HERAS^{AF}

Bachelor Thesis

Rich Client Platform: Reference Architecture

Department of Computer Science
University of Applied Sciences Rapperswil (HSR)
Spring Term 2009

Students: Tobias Forster, Ylli Sylejmani
Examiner: Wolfgang Giersche
Coaches: René Eggenschwiler
Expert: Dominik auf der Maur

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Disclaimer

We ensure with this disclaimer that this thesis was created by ourselves. We did not use any additional resources then the ones mentioned in the bibliography in the appendix.

The figures we used for this thesis were created by ourselves or are attached with a reference to the original author.

Each copied text phrase has an attached reference to the original text.

This thesis was not given, in this or in any similar form, to an examination board.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS^{AF}

Acknowledgement

First we would like to thank Wolfgang Giersche and René Eggenschwiler for their contribution to our good work and collaboration. Thank you for taking your time to give us a small introduction into OSGi, for your professional assistance and support at any time.

We also would like to thank the HERAS^{AF} developers, especially Florian Huonder, for their will to provide us information at any time and for their professional advices.

Additionally, we would like to thank everyone who supported us writing this bachelor thesis.

Abstract

Overview

In today's times the most applications are implemented using different user interface technologies, like Java Swing, C#.NET WinForms or further ones. This results in an increased development and maintenance effort. The use of a common platform is therefore an obvious topic, but brings some challenges. Beside the management and its continuous development, also problems, like missing flexibility and extensibility, are important issues.

Therefore a solution would be desirable, which is characterized through a modular structure and modern software engineering principles.

Goals

The goals for this bachelor thesis are for this reason the development of a reference architecture. This covers especially the following points:

- Development of a modular and service-oriented reference architecture based on OSGi and using Eclipse RCP for the user interface.
- Utilization of the Spring Framework and the Spring Dynamic Modules for the simplification of the implementation and the gain of flexibility.
- Definition and setup of a development environment including an automatic build process, based on Maven.
- Definition of development processes and best practices for future developers.
- Documentation of the solutions, alternatives, design decisions and a detailed developer's guide in English.

Solution

The developed reference architecture allows a modular design and implementation of applications, supported through the service-oriented approach of the underlying OSGi framework. The provision of services allows the separation of the business logic from the platform-specific components. As a result, it is completely reusable.

The platform-specific parts and especially the user interface are based on the Eclipse Rich Client Platform. It is also built upon OSGi, why it follows the same principles of encapsulation and modularity. Through the default components, Eclipse RCP provides solutions for frequently occurring problems. This covers simple user interface elements such as buttons as well as complex components, like a navigator or graphical editors.

Through the use of the Spring Framework and the corresponding Spring Dynamic Modules, the wiring of the components and the whole applications is possible. Thereby, the OSGi characteristics are considered. This allows the registration and lookup of services in a declarative manner. Specific implementations of them can be omitted.

The whole solution is supplemented with an automatic build process. Maven is its base, embedded in the OSGi environment through Apache Felix's *Bundle Plug-in for Maven* and the Spring DM testing framework. The integration of an automatic product generation using Eclipse's PDE Build completes the process.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Table of Contents

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Part I: Introduction

1 Intended Audience

General

This bachelor thesis contains information about the developed Rich Client Platform reference architecture. It describes the structure of the modules, interfaces and classes, its design and instructs the developer how to configure and implement own solutions.

Additional information about the workflow of specific parts, such as modules, packages, classes and interfaces, is also provided. In order to allow a reconstruction of the chosen design, detailed descriptions about the decisions are included. Hints and further help topics make the development, extension and configuration of own implementations easier.

A developer should bring basic understanding of the technologies showed in the list below.

- Java 1.5
- OSGi Framework
- Eclipse technologies, such as PDE and RCP
- Spring Framework and Spring Dynamic Modules
- Maven2
- Ant
- TestNG and JUnit
- UML and common software engineering principles and patterns

2 Management Summary

2.1 Initial Status

Motivation

Nowadays, most applications are developed using different user interface technologies. The consequences of such a heterogeneous composition are a higher development and maintenance effort. In such cases, the specification of a common platform is obvious, but brings some challenges with it. Beside the management and the further development of the own platform, problems, like missing flexibility and extensibility, can occur.

Existing work

The development of this bachelor thesis is not based on any already existing work. The result of it is created from scratch with own research and development effort aiming to achieve the goals.

Goals

Based on the mentioned challenges, the following goals have been set for this bachelor thesis:

- Development of a modular Reference Architecture using OSGi and Eclipse RCP.
- Use of Spring Dynamic Modules for the simplification of the implementation and the increase of the flexibility.

- Specification of a development environment which is optimally supported by an automatic build process based on Maven.
- Definition of development processes and best practices.
- Documentation of the solution as well as the creation of a detailed developer's guide in English.

2.2 Realization

Involved persons

Person	Function
Wolfgang Giersche	Responsible tutor
Dominik Auf der Maur	Expert
René Eggenschwiler	Coach
Tobias Forster	Project Manager, Programmer
Ylli Sylejmani	Lead programmer, Quality manager

The functions assigned to Tobias Forster and Ylli Sylejmani show primarily the accountability and not the responsibility. Both are responsible for the successful realization of this bachelor thesis.

Familiarization

To become familiar with the project and the used technologies, a detailed research on Eclipse RCP, OSGi and Spring Dynamic Modules had to be done.

Procedure

This bachelor thesis was realized based on an agile process, which mostly corresponds with Scrum *[Scrum]*. The iterations were variable, depending on the size and complexity of the tasks. Generally, they can be grouped into the following six phases.

Phase 1: RCP Basics and Technology Know-how

This phase lasted two weeks and consisted of the research on the new technologies. This especially includes Eclipse RCP, OSGi and Spring Dynamic Modules. Further, the setup of the infrastructure was an issue.

Phase 2: Advanced RCP, Spring DM and Automatic Build

In this five weeks lasting phase, RCP and OSGi tutorial applications were created beside the analysis and the development of an extensive automatic build environment. This also included the evaluation of appropriate Maven plugins as well as the documentation of the results.

Phase 3: Analysis of the Text Editor and GEF/GMF

This phase lasted three weeks and contained the analysis of the text editor as well as first implementation attempts. Further, an evaluation about the possible technologies for the graphical editor was done. Beside the research, first example implementations were created. The gained experiences were documented.

Phase 4: Implementation of the Text Editor and GEF/GMF

Based on the results of the analysis phase, the development of an integrated

RCP application including both editor types was the intent of this four weeks lasting phase. Additionally, the design of the architecture has been validated to be stable and functional. Here, as well, gained experiences and results were documented.

Phase 5: Refactoring and Advanced Topics

This phase lasted one week. The tutorial applications were refactored and modifications to the architecture were applied, where required. The documentation of the experiences and results was finished.

Phase 6: Finalization and Documentation

This two weeks lasting phase included clean-up tasks and the finalization of the whole documentation. The quality of the code was checked again while verifying and updating the javadoc of the tutorial applications.

2.3 Achieved Goals

Results

Automatic Build

As a given goal for this bachelor thesis, we made a proof of concept on an automatic build solution considering OSGi characteristics.

The result of this part is an integrated build environment based on Maven. Through the use of Apache Felix's Bundle Plug-in for Maven OSGi specific configurations and implementations are supported. This covers beside the manifest generation also the creation of valid OSGi Bundle Archives.

Complementary, the execution of unit and integration tests inside of the OSGi environment is possible with the use of the Spring DM Testing Framework. It encapsulates the tests with pre- and post-processing. These cover amongst other steps, the generation of an on-the-fly test bundle, the setup and startup of an appropriate OSGi environment and the cleanup and teardown at the end.

The whole build process is completed with the automatic generation of deliverable RCP products. This has been achieved through the integration of Eclipse's PDE Build.

Reference Architecture

The development of a reference architecture is the main goal of this bachelor thesis. It consist of the following parts:

- Modular application architecture based on SOA principles
- Integration of RCP components, such as a Text Editor, Graphical Editor, Problems View and furthers.
- Definition of OSGi Services
- Synchronization of the different components in a flexible and encapsulated structure
- Wiring of the whole application using Spring, Spring DM and Spring Utilities
- Design of unit and integration tests for the verification of the application

Documentation

The documentation is the third part of the bachelor thesis. It consists on the one

hand of a detailed analysis and the description of the solution. On the other hand it provides the definition of development processes and instructions for developers how to implement custom applications based on this reference architecture. The documentation includes additionally best practices and recommendations for this purpose.

Personal experience

During the bachelor thesis we made a lot of new personal experiences. These are shown in the list below:

- Working with an agile software development process and the appropriate tools
- Using software engineering principles and best practices practically
- Extensive usage of various design patterns, like e.g. whiteboard, strategy, command, extension interface and others
- Development of enterprise applications
- Writing a detailed technical report and developer's guide in English
- Teamwork and organization in an agile software engineering process
- Comprehensive research on new technologies, like OSGi, Eclipse RCP, Spring Dynamic Modules and others

2.4 Outlook

General

In this chapter further extensions and changes of the developed reference architecture are described in an aggregated form.

Technologies

Consideration of coming technologies

In this bachelor thesis, the newest available tools and libraries were used. In future implementations, though, coming technologies should be considered. Developers especially should keep an eye on the further development of the tools used in the automatic build solution as there is still room for improvement in terms of usability, simplicity and the integration of Eclipse technologies.

Continuous upgrade of used frameworks and tools

The used frameworks and tools in the developed reference architecture should be updated continuously. Through this, the developer can benefit from further implementations and thus from simplifications and more stable tools. Beside the Eclipse framework and Spring, also the tools used in the automatic build solution should be upgraded regularly.

Automatic Build

Monitoring the progress of the Tycho Plug-in

The Tycho plug-in integrates the common steps for the creation of OSGi bundles, like e.g. generation of the manifest, and a complete creation of a product using Eclipse technologies. However, the Tycho project is currently still in the development phase. In the future, the progress of it should be monitored.

Improvement of the PDE Build

In the automatic build solution, especially the PDE build leaves room for improvement. In the current solution, the Maven AntRun plug-in and Ant scripts

are used to accomplish this task. In the future, though, coming technologies and new versions of used libraries and tools should be considered in order to develop a better integration of the PDE build in the automatic build process.

Reference Architecture

Integration of extended services

In the future, the integration of extended services should be considered. An extended service can be e.g. a web-service and/or a service providing advanced access to a relational database. Such an extension broadens the horizon of the applications implemented based on this reference architecture.

Implementation and integration of further RCP components

The developed reference architecture in this bachelor thesis gives the base for the implementation of sophisticated rich client applications. In the future, the implementation and integration of further RCP components should be considered.

Switch to TestNG for the integration tests

In the current version, Spring DM only supports JUnit as a testing framework for the development of integration tests. According to Spring, an integration of the TestNG testing framework is planned in the future. Therefore, the further development of the Spring DM project should be monitored. A switch to TestNG enables the creation of advanced tests through the use of modern technologies.

Documentation

Continuous update of the documentation

The documentation is an important part of this bachelor thesis. As already mentioned before, the used technologies will be changing in the future and providing more simplifications and further functionality. Therefore and in order to simplify the use of the developed reference architecture, the documentation should be updated continuously, as well.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Part II: Thesis & Analysis

3 Thesis

Overview

In this chapter we explain the thesis in and its detail. This covers in particular the vision and goals to get an impression about the scope. The underlying technological base is mentioned in relation to the goals.

3.1 Vision

Vision

The vision of this bachelor thesis is the development of reference architecture based on an OSGi environment. Its modular structure allows the design of extendable and flexible applications.

Through the Eclipse Rich Client Platform, which is built on the OSGi framework, exactly this concept can be used for the rich client development. Thereby it is possible to implement a client in more or less independent components, which can be composed afterwards to a whole application.

The reference architecture should consist of an RCP application combining a text editor with a corresponding parser to convert its content to a domain model. The domain model should be further displayed and edited in a graphical editor.

Additionally, the whole application development should be supported with an automatic build environment considering the OSGi characteristics, executing unit and integration tests as well as building deliverable products.

Its detailed documentation allows the development of individual RCP applications in a straightforward way. It considers state-of-the-art software engineering principles, like DRY[1] or KISS[2], as well as OSGi and Eclipse best practices.

Vision diagram

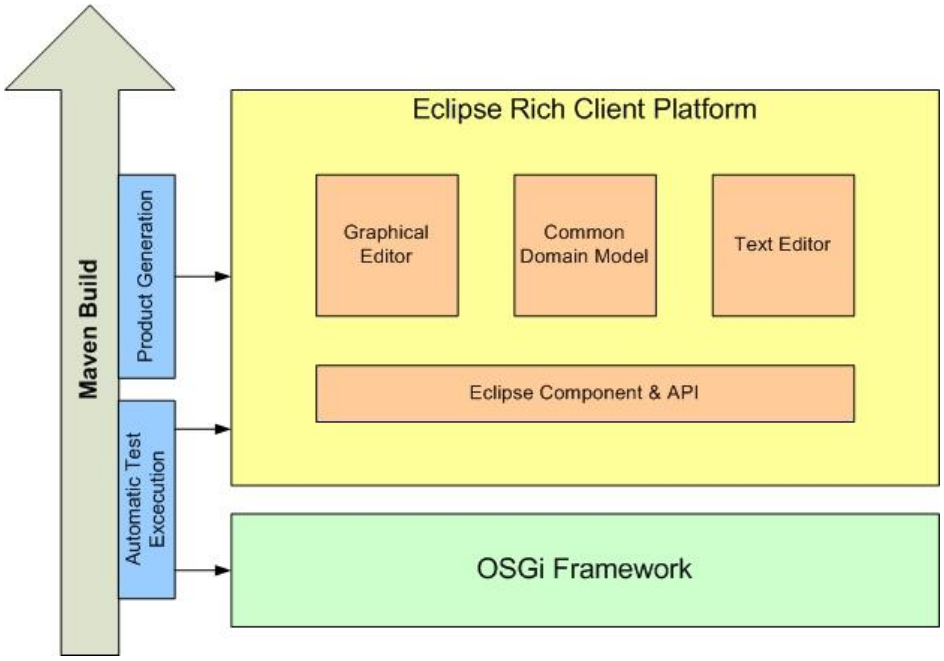

Figure 1: Schematically visualization of the reference architecture vision

[1] DRY: Don't repeat yourself
[2] KISS: Keep it simple, stupid

Motivation

The motivation to do this thesis is mainly the possibilities through the new technologies and especially OSGi and the Spring Framework. They help to reach popular software engineering principles, like high cohesion and low coupling, in an easier way. Also other problems, like classpath conflicts through the use of the same libraries in different versions, can be solved.

The modular structure of OSGi allows additionally the implementation of applications following the SOA principles inside the Java Virtual Machine. Therefore a better encapsulation is possible and consequently a high flexibility can be provided. The exchange of components through substitutes is easier than ever before.

This is supplemented with the Eclipse Rich Client Platform, which is build upon the OSGi framework. Therefore the implementation of clients is not only supported with default components but also with the possibilities through the underlying OSGi framework.

Additionally, the integration of the whole development process, using the mentioned technologies and principles, in an automatic build environment, allows the reduction of manual work. Maven and its plug-ins form the base for this purpose.

3.2 Goals

Introduction

The goal of this bachelor thesis is mainly the development of a reference architecture using the Eclipse Rich Client Platform. This covers in particular a detailed documentation about the design decisions, solution descriptions as well as guides and best practices for an own development. These descriptions must be broadly based on references and own experiences.

Goals

Integral parts of this reference architecture are the following topics:

- Development of an automatic build environment with Maven considering the OSGi specific characteristics.
- Design and implementation of unit and integration tests for the verification of the different bundles and plug-ins as well as its execution as part of the automatic build environment.
- Choice of an appropriate architecture for the development of RCP applications, considering state-of-the-art engineering principles and best practices.
- Development of a text editor component with highlighting, context menu, content assistance and annotation functionalities. A parser should convert the displayed text into a domain model structure.
- Development of a graphical editor component for the visualization and manipulation of the domain model.
- Design of a common domain model for the whole RCP application, which is provided and managed centrally. A mechanism should allow the notification of dependent or registered components on changes.
- Configuration and integration of Eclipse components in RCP applica-

tions.

- Preparation of tutorials as a guide on how to implement parts of the reference architecture.

Documentation

Each of the mentioned topics covers also the documentation of the solution, which should consist of design decisions and the manner how the components were implemented. A guide should explain how an own implementation can be achieved based on recommendations and further advices. Alternative solutions should be mentioned as well and how they would be developed.

3.3 Technologies

Overview

The whole thesis is based on the specific technologies, like OSGi, Eclipse RCP and Spring Dynamic Modules. They are an integral part and must be used to fulfill the given requirements. In this chapter we explain these technologies and their role and purpose in this thesis.

3.3.1 OSGi

Introduction

OSGi stands for Open Services Gateway initiative. It is a software framework simplifying the encapsulation of applications and services modules, using a component model, and its management. It is built on a Java-based runtime environment.

An important characteristic of the OSGi framework is the possibility of dynamic and controlled installation, startup, stop and deinstallation of service bundles at runtime, as shown in **Figure 2**. Its model allows therefore the parallel execution of independent and modular applications at the same on the same Virtual Machine. The dependencies between the bundles, how the modules are called, are resolved automatically combined with an integrated version management.

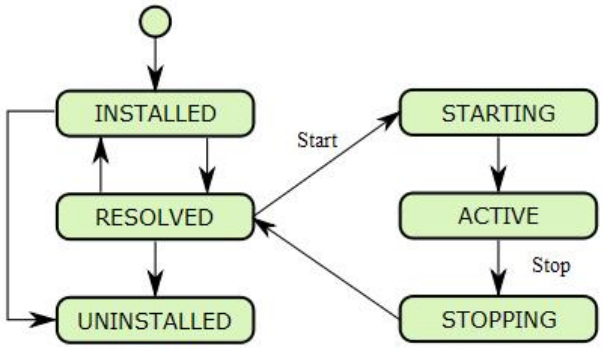

Figure 2: OSGi bundle lifecycle (Source: Wikipedia)

Originally designed for mobile and embedded systems, the OSGi framework is today widely used, for example as the base of Eclipse RCP or application servers.

API definition

The extended manifest file allows further the definition of a bundle API. This means the bundle developer can decide which packages should be available to for the others and which ones not. For this purpose the OSGi framework provides corresponding manifest headers, like *Export-Package*, *Private-Package* and *Import-Package*.

Services

All this is supplemented with the possibility of service definitions. Bundles can implement simple POJO classes and register them as services in the OSGI ServiceRegistry using a predefined interface.

Another bundle which requires such a service can do a look up at the ServiceRegistry using the same interface. What an instance it gets is controlled by OSGi respectively its ServiceRegistry. But it has to satisfy the lookup interface.

This allows the development of applications in a SOA-like manner with simple POJOs inside the Java Virtual Machine, like shown in **Figure 3**.

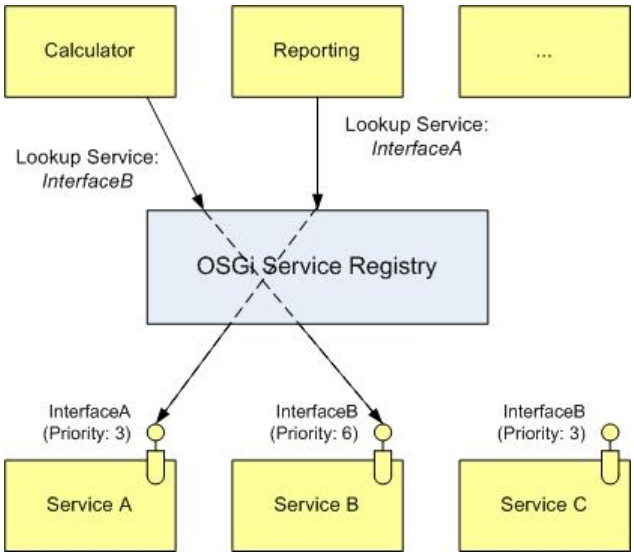

Figure 3: Schematical visualization of the reference architecture vision

Further information

Further information about the OSGi framework can be found in *[OSGiCore]* and *[Wütherich08]* as well as on the official OSGi website at http://www.osgi.org/.

3.3.2 Rich Client Platform

Introduction

Eclipse's Rich Client Platform is a framework for the development of rich client applications. It is based on the OSGi framework which allows its modularization and an easy extension with further features as own bundles.

Features

The framework consists of several components and features which provide the basic and common functionalities of a rich client application. They can be configured and used for the development of own applications to allow a faster implementation. So their recurring implementation or an own management of a basic framework is not necessary anymore.

Extension Points — This is supplemented with the extension point mechanism, which allows the extension and configuration of the existing framework code base. This is necessary for those parts where the framework implements a general behavior whose adaption with inheritance is not possible or not recommended.

Further information — Further information about Eclipse RCP can be found in *[Daum2008]* as well as on the official Eclipse website at http://www.eclipse.org/rcp/.

3.3.3 Spring Dynamic Modules

Introduction — The Spring Dynamic Modules is a subproject of the Spring Framework. It acts as a bridge between the conventional Spring Framework and the OSGi environment. The basic idea behind it is the provision of Spring as bundles, where the dynamic characteristics of OSGi should be considered and used.

Application context — For this reason each bundle has its own application context. It is totally independent from the ones in other bundles. This allows an individual wiring of the classes.

Services — Based on this integration, Spring also allows the registration of Spring beans as OSGi services as well as their lookup in a declarative manner. Services can therefore be injected and used as POJOs in the classes. Through the lifecycle support Spring DM provides an automatic refresh of the service instances, because they can be installed and deinstalled dynamically.

Further information — Further information about the Spring Dynamic Modules can be found in [on the official Spring website at http://www.springsource.org/osgi.

3.3.4 Maven

Introduction — Maven is a build management tool for building and managing Java applications. It tries to automate as much steps as possible for the whole software development cycle to reduce manual steps.

POM — Following the given standards only a few configurations for the steps must be done. Maven provides for this purpose the Project Object Model, or pom for short. It must be located in the projects of the application and contains configurations like names, version, properties, as well as dependencies.

Dependency management — This allows the management of whole dependency trees through Maven, which means that Maven resolves the dependencies of projects and Java archives automatically through the entries in the pom and gets them. Therefore it isn't necessary anymore to get the dependencies of the dependencies on your own.

Plug-ins — As an additional feature, Maven can be extended and adapted easily through its plug-in structure. Therefore it is possible to include Ant tasks, get access to a source management tool, generate documentations or consider special charac-

teristics of the application which should be built.

Bundle plug-ins

This allows the implementation and usage of plug-ins considering the special requirements of OSGi with the manifest and product generation and further more against a defined target platform, as shown in **Figure 4** below.

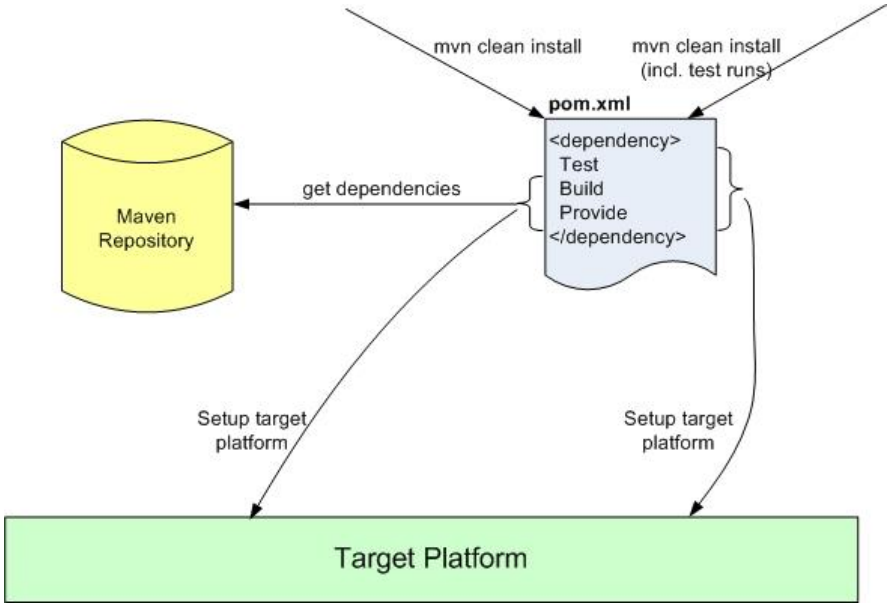

Figure 4: Schematically Maven workflow for building applications and setup the target platform

Further information

Further information about Maven can be found on the official Apache Maven website at http://maven.apache.org/.

4 Analysis

Overview

In this chapter we analyze the general and global topics of the bachelor thesis. They allow in the further steps the definition of concrete requirements, making decisions for concrete solutions and further more. But no decisions will be made in this chapter.

Additionally, the analysis descriptions should only provide ideas and lead to questions, which must be decided during this thesis as part of it. As the result, the scope of the bachelor thesis should be defined.

Detailed descriptions, analyses and decisions to specific topics are handled in chapter 6.

4.1 Reference Architecture

Intention

The development of a reference architecture is the main goal of the bachelor thesis. It consists of several topics, which can be handled independently. So the idea of the reference architecture itself is more the combination of different topics and the way to do it, than the detailed analysis of them.

This allows here an analysis of the global view as a first step. Its results can be used as a base in the further chapters for a more focused analysis. So these components consider from the very beginning the global requirements.

4.1.1 Application architecture

Overview

The first of these global topics is the architecture of the application. It is its backbone which has to fulfill various expectations and requirements like functional and non-functional ones.

Architecture

The application architecture must be flexible enough to allow a generic adoption for different purposes and aims. On the other hand, the flexibility should not lead to fragile or too complex structures, because their maintainability would suffer.

For this reason it must be possible to do changes and extensions in an easy manner to fulfill a wide range of requirements. This supposes that they have to be done only selective. Otherwise changes are needed all over the application which corresponds with the *Shotgun Surgery* smell *[Smells]*.

Additionally, the choice of the architecture type has direct impact on the way how the classes and interfaces can be bundled in own Java archives.

Alternatives

Alternative 1: Layering architecture

In a layer architecture each level has its responsibilities. For example the UI layer is responsible for the graphical visualization and to accept the user interactions, whereas the subjacent layer would be responsible to handle them. Such an architecture is very useful if the whole layers should be exchangeable through substitutes without changes in the remaining parts. This is especially interesting for platform-specific implementations like a user interface or something else.

Advantage	Disadvantage
• Allows the exchange of whole layers without changes in the remaining parts. • Implementations with the same responsibilities are grouped.	• An exchange of whole layers requires the definition of clear interfaces between them. Otherwise all connections between them must be adapted. • Risk of the monolithic layers through a missing separation of the implementations belonging to different components.

Alternative 2: Slicing architecture

A different approach is an architecture with slices, which consists of several parts like the user interface, interaction handling, database access and further more. Typically a slice represents a feature or a component which contains the required implementations through all layers. Such an architecture is very useful for applications which are composed of different features, which can be added respectively removed easily.

Advantage	Disadvantage
• Allows an easy addition and removal of features as own components.	• Different components may contain similar implementations, because they cannot depend on other components. • Risk of monolithic slices with missing separations between the responsibilities.

Alternative 3: Layer-slice architecture

A combination of the layered and sliced architecture is a further approach. It allows the exchange of platform-specific and the use of common implementations through its layering. Additionally the slicing gives the possibility to add and remove components in an easy manner.

Advantage	Disadvantage
• It combines the advantages of both solutions, allowing the exchange of layers as well as the easy additional and removal of components. • Platform-dependent implementations can be encapsulated and exchanged easily as well as common parts.	• The combination of both solutions leads to a dilemma, where to use layers and where slices.

4.1.2 Exchangeability of components

Overview

An important requirement of a reference architecture is its flexibility. This covers especially the exchangeability of components and implementations through substitutes. Such an architecture would allow the developers to change for example the business logic, but without adjustments in the using component.

Alternatives

Alternative 1: OSGi Services

The OSGi framework allows the use of services. This means classes can be registered as services in its *ServiceRegistry*, from where they can be resolved and used. For this purpose an interface definition is required, which describes the available methods with its parameters and return values.

This feature allows a very flexible implementation of application components, because they can be exchanged and replaced in a simple manner. The clients of the services have only dependencies to the service interface and can be implemented against them. What concrete implementation will be used doesn't matter.

Advantage	Disadvantage
• The implementation of a client against an interface is possible without taking care what implementation will be provided. • Application components can be exchanged in a simple manner, because not direct dependencies exist. • The service and its clients run all in different operation contexts with own classpaths. Therefore they can use different libraries without conflicts. • The Spring Dynamic Modules allows the registration of services in a declarative way. This allows the implementation of services as POJOs.	• Depending on the requirement if a service is mandatory to allow the use of a component, it is possible that the application will block until a lookup can be satisfied. • Can be used only in an OSGi based environment.

Alternative 2: Manual component encapsulation

A similar approach is the implementation of the component encapsulation as an own component. For this purpose a registry element would be necessary, otherwise the registration and resolution of services is not possible.

Advantage	Disadvantage
• The clients can be implemented fully against interfaces without taking care what concrete implementation will be	• Depending on the requirement if a service is mandatory to allow the use of a component, it is possible that the application will block until a

available. • Application components can be exchanged in a simple manner. • Can be used either in an OSGi environment or in a conventional one.	lookup can be satisfied. • The service and its client run in the same operation context. Therefore conflicts caused by incompatible libraries can occur.

4.1.3 Common Parts

Overview

A frequently challenge which occurred typically, is to provide common parts for all components of the application. An example is a jointly used domain model, created by an input mask, visualized in a diagram and manipulate in a table component. All of them need the same domain model instances.

Beside an own implementation of a global provider, which takes this responsibility, OSGi allows also the use of OSGi services for this purpose. Through a proper configuration, a single instance can be provided for all of its clients.

Alternatives

Alternative 1: Global provider

A first approach to handle this challenge is the implementation of a global provider. It takes the responsibility to provide its clients with the required data and instances.

Advantage	Disadvantage
• Allows the jointly use of central components like a domain model. • Its use is not limited on an OSGi environment.	• The synchronization of parallel accesses must be implemented to avoid problems. • An own mechanisms must be implemented to notify dependent components about changes. Therefore they must be managed in the global provider. • The components have a direct dependency to the global provider.

Alternative 2: Single OSGi Service

Another approach is the use of the OSGi ServiceRegistry for this purpose. To its responsibilities belongs the management of central and jointly used parts. So it's obvious to use it not only to provide services but also jointly used parts, like a domain model. Through a proper configuration of the services it is possible get a single instance of it for the whole application.

Advantage	Disadvantage
• Allow the jointly use of central components through a proper	• It has the same disadvantages like conventional OSGi managed ser-

configuration of services, to be “singletons”.

- Through a declarative configuration with Spring DM the components don’t need further dependencies.
- Together with the Spring DM an easy notification mechanism using the Whiteboard Pattern *[Kriens04]* can be provided.
- It has additionally all the advantages like conventional OSGi managed services.

vices.

- The synchronization of parallel accesses must be implementation to avoid problems.
- It is limited to OSGi based environments

4.1.4 Component integration

Overview

Another important challenge is the integration of further components as well as their removal. This should be possible without a huge change effort, but depends directly on the chosen architecture (see chapter 4.1.1).

Extension point

Eclipse RCP provides for this purpose the extension point mechanism. It allows the configuration and extension of existing parts with specific functionality. Changes in the existing code base are not required, whereas the classes for the new functionality have to correspond to a specific interface. Otherwise, a common handling of the new features is not possible.

4.2 Automatic Build Environment

Intention

Another goal of this bachelor thesis is the automation of the development process. This consists mainly of a reduction of manual steps for resolving dependencies, building projects and further more. It simplifies the daily work to allow a concentration on the real challenges and problems.

4.2.1 Automatic Build of different projects of an application

Overview

An important step in the development is the compilation of applications, which are distributed into several projects. Typically changes in one project have impacts in another one if they are built. To check the build of a whole application all is projects should be built together, not one after the other. This helps to reduce old dependencies, because no project is forgotten.

Maven

Maven provides this feature through the definition of a parent pom. In its content the subprojects which belong to the projects must be declared. Otherwise they are not known and cannot be considered for the build process.

Afterwards, the specific build steps for the parent pom will be invoked and then

for the declared subprojects. These again can also contain declarations of subprojects.

Parent project

For the location of the parent pom generally two solutions exists. In the first one, an own project is created, for example in Eclipse, which contains the parent pom. It allows a common handling of all projects. For the declaration of the subprojects the folder structure must be considered.

Folder hierarchy

Another approach is the definition of a folder hierarchy where in the parent folder beside the parent pom also the subprojects are located. In such a case the parent pom cannot be managed as an own project, because it would lead to conflicts with its subprojects. But the remaining configurations and uses are very similar to the approach with an own project for the parent pom.

4.2.2 Consideration of OSGi characteristics

Overview

The use of the automatic build in an OSGi environment must consider its characteristics for an optimal support. Otherwise several steps or configurations have to be made manually, which leads to a higher risk of errors. Therefore, they should be avoided as much as possible.

Maven plug-ins

For this purpose several Maven plug-ins exist, whose aim is to provide an optimal support. This should be achieved through the automation of configuration and build steps, which are specific for an OSGi environment. Some of them are more experimental nature than really useful for the application development, like the OSGi Bundle *[MVNOSGi]* of the ETH Zurich. Others would bring one of the widest integration steps including the specific product generation, like the Tycho plug-in *[Tycho]*, but are still under construction.

Apache Felix

Therefore it is required to get an existing plug-in which provides the quality for a use in a development environment, like the one of the Apache Felix project. But its functionality is mainly limited on the generation of a manifest file and the integration of further resources into the Java archive.

4.2.3 Generation of deliverable products

Overview

To provide a huge simplification, the automatic build environment should also support the generation of deliverable and executable products. Typically this is not an issue, because in conventional applications only Java archives have to be generated, which is already provided by build management tools, like Maven.

In some cases this is not enough, because further resources and files must be generated. An example is an own client application based on Eclipse RCP and OSGi. Here also the generation of a whole folder structure with several Java archives and further parts is required including an *.exe* file as well.

PDE Build

For this purpose Eclipse provides the PDE Build, a plug-in whose core responsibility is the generation of executable products. It can be used directly as a feature in Eclipse as well as from outside. In this case it must be configured through

passing parameters.

Maven plug-ins

That's the reason why the development of Maven plug-ins, which handles this issue is possible. They encapsulate the PDE Build invocation and can therefore be included in the existing build process to supplement it with the product generation. An already existing plug-in is the Eclipse PDE Maven Plug-in *[MVNPDE]*. Also the Tycho plug-in *[Tycho]* which is under construction plans to support the product generation.

But they have also huge disadvantages like more or less rigid restriction. Some of them are limiting in a manner, covering the prescription of a specific folder structure, they cannot be used.

Ant file

For this reason, another possibility is the invocation of the PDE Build from an Ant file. Since the PDE Build is implemented also based on Ant files, this is also an obvious way for the integration of the product generation. Unfortunately in this case is the extra integration of the Ant invocation in the Maven build process. But it is possible through the AntRun plug-in.

A detailed comparison of possible solutions for the product generation can be found in chapter 6.1.1.3.1.

4.2.4 Continuous integration

Overview

The automatic build environment can also be used on build servers, like Bamboo, for the continuous integration. Typical purposes are to run a build after changes are checked-in in the version control tool or at a specific time, like e.g. nightly builds.

Infrastructure

But for such a continuous integration also the required infrastructure must be available. This is an important issue for the generation of deliverable products, like for the PDE Build in an OSGi environment.

4.3 Client application

Intention

An important part of this bachelor thesis is also the implementation and configuration of the client components using Eclipse RCP. Especially their integration in the existing application in a proper way is one of the main challenges, because RCP requires more or less a considerable amount of dependencies. They should be kept as low as possible.

4.3.1 Encapsulation of components

Overview

The implementation of an RCP application consists typically of several components. Some of them interact with each other in a way that the result of one component is the input for another one. Generally, the challenge is to keep the direct dependencies as low as possible. This allows a replacement or a removal

of components without changes in the remaining parts.

Extension point

Eclipse RCP provides for this purpose the extension point concept. It allows the implementation of components against them using defined interfaces or abstract classes. Thus it does not have to take care about the concrete implementations. See chapter 6.3.1 for additional information on this topic.

Wiring with Spring

Also the wiring of components using Spring supports the encapsulation, following the same concept as the extension points. Through the implementation against clear defined interfaces, the components have a low coupling, which is intended in this way. For additional information see chapter 4.3.7.

4.3.2 Notification mechanism

Overview

An important concept supporting the encapsulation of components is the notification mechanism. Its responsibility is the notification of dependent and/or registered classes and components about changes. The way how it is implemented may support the encapsulation directly.

Alternatives

Alternative 1: Own implementation

A first way to use such a mechanism is to implement an own one. Typically it consists of a manager holding the registered parts and an interface or abstract class representing the registered elements.

Advantage	Disadvantage
• The notification mechanism can be designed with considerations of the specific usage. • Typically it is a simple implementation. • Through the location in a utility package or bundle the components using it have only dependencies to them and not to the classes implementing it.	• For this problem Java and Eclipse RCP provides already solutions. Therefore an own implementation would be duplicate. • The notification mechanism must be implemented in a generic way to allow a wide range of application. • Synchronization mechanisms must be considered in methods where concurrency problems may occur.

Alternative 2: Java's Observer implementation

A very popular implementation of a notification mechanism is Java's Observer. It consists of a class acting as the manager `Observable`, which holds the registered elements and provides the notification algorithm itself, and an interface `Observer` for the implementation of the elements which want to be notified.

Advantage	Disadvantage
• It is a generic default implementation provided by Java, which has proven as stable	• The names of the class and the interface are a little confusing.

and useful. • Its configuration and use is very simple. • Through the generic structure it can be used in a wide range of application. • Concurrency problems in the observer manager are already considered.	• The considerations of specific behavior must be implemented on its own.

Alternative 3: JavaBeans property change

A solution especially designed for Java beans is its property change mechanism. Therefore it is very similar to Java's Observer implementation, but allows passing an old and the new value of a property for the notification.

It also consists of a class acting as the manager `PropertyChangeSupport` and an interface `PropertyChangeListener`. The manager holds the registered elements and provides the notification algorithm itself, whereas the interface is used for the implementation of the elements which want to be notified.

Advantage	Disadvantage
• It is a specific implementation for the use in Java beans. It has proven as stable and useful. • Its configuration and use is very simple. • Sensitive methods, which are at risk for concurrency problems, are implemented with synchronization mechanisms. So they don't have to be considered.	• It is designed for the use in Java beans. Therefore its application is limited.

Alternative 4: Eclipse's Data Binding

Eclipse RCP itself provides also a kind of notification mechanism called Data Binding. Its main purpose it the synchronization of two attributes in two different classes. A typical application area is the automatic synchronization of a user interface field with the underlying domain model element. Therefore it is also able to carry out conversions and checks.

Advantage	Disadvantage
• It is a specific implementation for the synchronization of attributes or classes in an automatic manner. • It includes also checks and conversion steps to allow a	• Its main purpose is the synchronization of UI elements with the underlying domain model attribute. Therefore its application area is very limited. • The configuration of the Data Bind-

very generic synchronization.	ing is very complex and cumbersome. • The elements at both ends must exist, to allow the use of Eclipse's Data Binding. Otherwise a different mechanism must be used.

Alternative 5: Whiteboard Pattern through OSGi services

Another notification mechanism often used in the OSGi environment is the Whiteboard Pattern *[Kriens04]*. The classes interested in the notification about changes don't have to register on the service itself. They implement a given interface and register themselves as services on the OSGi ServiceRegistry instead. If changes in the service occur, it looks up the "listener services" and notifies them. So the management of the registered elements is done by the OSGi ServiceRegistry.

Advantage	Disadvantage
• It is a very flexible concept for the use in an OSGi environment. • The management of the registered classes as services is handled by the OSGi ServiceRegistry. This simplifies its use. • Through the use of the OSGi ServiceRegistry no dependency to the concrete service implementation is required.	• Its use is limited to the OSGi environment. Therefore it cannot be used outside.

4.3.3 Configurations of components

Overview

The configuration and adaptation of the components is an important issue for the development of RCP applications. It is an integral part of several RCP components. Otherwise a general implementation of them would not be possible or only hard to achieve.

Extension point

For this purpose Eclipse RCP provides the extension point concept, as already mentioned above. It allows an easy configuration and adaption of the components based on the specific requirements through an inversion of control. This means they will be invoked through the general implementations and not vice versa. Therefore, a considerable part of the required implementation can be replaced through them.

Programmatic

Nevertheless, also a programmatic configuration of the RCP components is possible. Compared with the extension points they are less flexible, because they cause adjustments in the code if they should be changed. But in some cases they are the only way, because the extension points are not as fine-grained

as required to replace the programmatic approach completely.

Spring

To simplify these configurations and to gain flexibility with their handling, the Spring Framework with its Dependency Injection feature can be used. For this purpose it would be necessary to define a common interface and to implement the configurations using it, following the Strategy Pattern *[GoF1995]*.

4.3.4 Widgets

Overview

Eclipse RCP supports the implementation of rich clients with several widgets and components. This covers very simple ones, like buttons and labels, as well as complex components, like graphical or text editors.

Intended use

For the development of the reference architecture, we intend to use the following components:

- Simple widgets like buttons, labels, combo boxes, dialogs and further more.
- Text editor component and its configurations, covering syntax highlighting, model reconciliation, outline view, context menu and further more.
- Graphical editor component and its features, covering outline view, palette, action bars, direct edit and further more.

4.3.5 Text editor

Overview

The text editor is a powerful RCP component, which provides several features supporting the user and is highly configurable. Therefore it allows the implementation from simple text viewers to complex editors, similar to the Java Source editor in the Eclipse IDE.

Integration

For the creation of the text editor and its simultaneous integration in the remaining application, Eclipse RCP provides the extension point `org.eclipse.ui.editors`.

It allows additionally general configurations like the file extension binding, to define for which files it should be opened. Also icon and action bar contributors can be set.

Configuration & Feature

Different the detailed configurations, which must be done programmatically. The reason for this restriction is mainly the granularity of them and the fact they are also features. Additionally they vary greatly between the different implementations.

Possible features and configurations are the followings:

- *SourceViewer:* The source viewer allows the extension of the basic text editor with syntax highlighting, extended tooltips named annotation hovers and the content assistance. Also the integration of a parser through

a reconciler must be done here.

- *Document:* The document model contains the content of the text editor and allows its manipulation through various methods. Additionally the definition of partitions is possible for the division of different types of content, like e.g. the source code and comments in a Java class.
- *Annotations:* Additional model for the definition of extra information for a piece of text in a specific position, like e.g. labels for errors, warnings and the like.
- *Folding:* Allows the definition of text blocks which can be minimized and maximized. An example of folding is the folding structure for methods in the Java class editor of the Eclipse IDE.
- *Templates:* Templates are patterns with placeholders which can be used to write structured text.
- *Outline page:* Is a view providing an overview over the structure of the text editor's content. As an example, in a Java project in the Eclipse IDE it shows the class with its methods and fields.
- *Action bars:* Allows the programmatic composition and configuration of menus, toolbars, context menus and appropriate entries.
- *Markers:* Markers are descriptions of errors, warnings and further more on concrete resources in the workspace. In the text editor they cause the illustration of them through an additional annotation model with squiggly marks and icons.

4.3.6 Graphical editor

Overview

Also the graphical editor is a powerful RCP component, which provides several features supporting the user and is highly configurable. Therefore it allows the implementation from simple visualizers to complete graphical editors to handle of complex tasks.

Technologies

Eclipse provides for this purpose two frameworks with different scopes and implementation support.

The first and older one is GEF, the *Graphical Editing Framework*. It provides default and abstract classes for the implementation of simple visualizers as well as complex editors.

Different is the *Graphical Modeling Framework* or GMF for short. It is built upon the *Graphical Editing Framework* and supplemented with the *Eclipse Modeling Framework*. Through this combination GMF allows the generation of graphical editors using configuration models instead of its implementation. The creation of graphical editors is therefore easier and much faster to achieve.

A detailed description and comparison of the two frameworks can be found in chapter 6.3.8.1.

Integration

For the creation of the graphical editor and its simultaneous integration in the remaining application, Eclipse RCP provides the extension point `org. eclipse.ui.editors`.

It allows additionally general configurations like the file extension binding, to define for which files it should be opened. Also icon and action bar contributors can be set.

Configuration

Different are the detailed configurations, which must be done programmatically. The reason for this restriction is mainly the granularity of them and the fact they are also features. Additionally they vary greatly between the different implementations and generation.

Possible features and configurations are the followings:

- *Content:* The content is the core of the graphical editor, because it contains the data and defines the visualization based on a MVC[3] class structure.
- *Outline page:* Is a view providing an overview over the structure of the graphical editor's content.
- *Miniature view:* A minimized visualization providing an overview about the whole diagram and shows the currently visible part.
- *Palette:* The palette is a special toolbar for the graphical editor, which contains the tools for the creation of new diagram elements, is selection and further more.
- *Drag & Drop:* Is a mechanism to drag a tool from the palette and to drop it on the diagram to create a new element. It allows therefore an additional way to create new elements.
- *Action bars:* Allows the programmatic composition and configuration of menus, toolbars, context menus and appropriate entries.
- *Direct Edit:* The direct edit is a way how to change the content of a diagram element through simple clicks and double-clicks on it. Otherwise wizards or other edit dialogs would be necessary.
- *Key handlers:* Allow the registration of keystrokes for the application of manipulations on the diagram elements and the diagram itself. Provides therefore a must faster way to manipulate the diagram elements.
- *Zoom and scaling:* Is a mechanism which allows changing the scaling of the diagram to get an overview or to visualize details.
- *Rulers and guides:* Allow the definition of a grid on the diagram which helps the user with the alignment of elements.
- *Snap to grid:* Mechanism supporting the user with the alignment and positioning of the elements in the diagram.
- *Wizard:* The wizards allow a different way to change the information about diagram elements as well as their creation.
- *Property sheet:* A third way to visualize several information about a diagram elements and its change. In the Eclipse IDE it is often used to show the properties about all kind elements.
- *Copy & Paste:* Provides the functionality to copy, cut and paste diagram elements for an improved usability.
- *Animation:* A mechanism for the visualization of changes in the diagram which can be used for the implementation of arrange functions and fur-

[3] MVC: Model-View-Controller is a design Pattern for the implementation of user interfaces. *[MVC07]*

ther features.

To consider for all this features is the fact that the interactions on the diagram and its elements are catched by their controller elements and passed to edit policies which create command objects.

Domain model differences

Also to consider is the fact that the graphical editor requires a different respectively an extended domain model, because it must contain also information for the visualization, like locations, sizes and further more. Therefore in the fewest cases the general domain model can be used.

4.3.7 Composition of a component

Overview

A further issue for the developer is the composition of components. This means the wiring of the classes, especially if they should be exchangeable in an easy way.

A flexible solution allows an easy replacement of components or parts of it through substitutes as well as its extension with new or additional functionalities. The reduction of the time and testing effort is for this purpose an important goal.

Generally the Java programming language allows a fix wiring by hand, which should be avoided as possible, and the use of factories, whose responsibilities are the instantiation of concrete classes. The Spring Framework takes this idea and allows the wiring of the classes with internal factories and dependency injection.

Alternatives

Alternative 1: Wiring by hand

Wiring the classes and components by hand means the simple instantiation of the required classes with the new operator. This is done in the classes which uses those implementations.

Advantage	Disadvantage
• Simple to implement, because no additional configurations and implementations are required.	• It is a less flexible solution, which causes huge efforts if changes must be implemented. • The implementation is numb. For the replacement of specific implementations every time a code change is needed.

Alternative 2: Use of configurable factories

A different approach is the use of factories or even configurable factories for the instantiation of the concrete classes. They encapsulate the instantiation process. Through the definition of common interfaces the classes can be implemented in a more flexible manner.

Advantage	Disadvantage
• Allows the encapsulation of specific implementations using interfaces and abstract classes. Classes can be implemented against them. • The instantiation of the concrete implementations is encapsulated. • The use of configurable factories allows an easy exchange of specific implementations.	• Requires additional implementation efforts for the factories and their configuration.

Alternative 3: Using Spring and Spring DM for the dependency injection

A third approach is the injection of dependencies provided by the Spring Framework. This means the concrete classes, which are implemented using interfaces and abstract classes, are instantiated. Afterwards they can be injected as objects in required classes, which are implemented against the common interfaces.

Advantage	Disadvantage
• Allows the encapsulation of specific implementations using interfaces and abstract classes. Classes can be implemented against them. • The instantiation of the concrete implementations is encapsulated. • No additional factory classes are required, because they are part of the Spring Framework. • The configuration of the dependency injection is done in an XML file. Therefore no changes in the source code are required.	• Additional libraries and bundles are required to use the dependency injection feature of Spring.

4.4 Application utilities

Intention

For the development of applications especially the utilities are an important issue. The general functionalities provided by them which are not part of the business logic. That's why they are used frequently for various purposes like formatting, converting and further more.

4.4.1 Logging

Overview

Logging the different states of an application is an important feature for the developer itself. It allows him to retrace the processing in the application, the state changes and furthermore, if required. This is useful in cases where errors occur to understand what happened before.

For this purpose several logging frameworks are provided, with their individual advantages and disadvantages.

Alternatives

Alternative 1: Own logging algorithm

A possibility is the implementation of an own logging algorithm. It can be adapted and configured based on the specific demands and requirements.

Advantage	Disadvantage
• The logging facility can be designed on its own. • Specific requirements can be considered.	• Because of several existing logging frameworks, it would be a duplicate implementation.

Alternative 2: Eclipse's `ILog`

The Eclipse platform itself provides also a logging algorithm. It is based on the status handling, using the `ILog` instance of the plug-ins. It is already integrated and used in RCP. If an application starts, the logging is automatically used to log which plug-ins are loaded, which are missing and further more.

Advantage	Disadvantage
• Through the integration in the Rich Client Platform, no additional configurations are required. • It is used by in the platform plug-ins itself to log different states, especially errors. • The default directory for writing the log files is the working directory of the application. Often this configuration is good. • It is exclusively designed for the use in an OSGi environment. Therefore no compatibility problems exist. • Through the implementation of log listeners, additional behavior can be provided.	• For the use of the Eclipse logging facility the Eclipse base plug-ins are required. This leads to direct RCP dependencies from all bundles and plug-ins. The implementation of platform independent components is not possible anymore.

Alternative 3: Logging frameworks

Also a possibility is the use of well known logging frameworks like SLF4J, Log4J and others. They have proven to be useful and stable during the development of conventional Java applications. Through their wide range of configuration possibilities they can be adapted easily, for example to filter log entries.

Advantage	Disadvantage
• They are well-known and widely used in the Java application development. • Through the wide range of configuration possibilities they can be adapted in detail.	• Some of those frameworks, like Log4J, are not prepared for their use in an OSGi environment. Therefore they must be converted in a first step. • Therefore the management of these bundles is required. • Documentations how to use the logging frameworks in an OSGi environment are very rare or incomplete. • Some of the frameworks have special requirements, like the availability of their configuration in the classpath. Through the fact, each bundle has an own classpath, this can lead to duplicate configurations.

Alternative 4: OSGi Log Service

A different approach is the use of the Log service provided by the OSGi framework itself. Since it is part of the OSGi specification, it must be provided by its implementations.

It consists of the three parts, whereas only one must be implemented on its own. The Log service itself can be used to write log statements without changes and implementations. These statements are stored temporarily in the log manager, until they were read using the LogReader service. For this purpose a component must be implemented either using a polling algorithm to check regularly for new log statements or as a listener, which can be registered on the log manager.

Advantage	Disadvantage
• Since the Log service is part of the OSGi specification is must exist in each implementation. • Its open structure allows a flexible use of the Log and LogReader service. • How to use and configure it is well documented in the internet and several books, like in *[Wütherich2008]*.	• Its use requires a higher initial effort, compared with other logging frameworks and facilities. • Through • The use of a service for logging is more complex than simply invoking a method of another logging framework.

- Through the implementation of log listeners, it is possible to get notified easily about new log statements.
- Since it is part of the OSGi framework, no additional libraries and bundles are required.

4.5 Testing

Intention

An important issue for the development of each kind of application is its verification through test cases. Beside the manual tests through users, the implementation of such through the developer is an important verification in an earlier phase.

In this case it can respectively it must be distinguished between different kinds of test cases. The unit tests verify a selected and limited part of a component, whereas the integration tests verify especially the interaction between the components.

4.5.1 Integration in build process

Overview

The integration of the text execution in the build process is a useful feature to verify the application every time the build runs. This helps to keep the quality of the implementations on a high level, because the build fails if the tests were not successful. But it requires also the continuous maintenance of the test cases; otherwise such an integration makes no sense.

Execution

The execution of conventional test cases, which does not depend on a specific platform or something else, is very simple. Typically they are unit tests verifying only an isolated part of a component. Maven uses for this purpose its default plug-in called Surfire. It executes the test cases, collects the return values and stops the build process if one or more failed.

4.5.2 Integration tests

Overview

From the site of the execution, more complex test cases which depend on a specific platform or require special libraries, it is very similar. Maven can also use its Surfire plug-in. But it is needed to do the required pre- and post-processing for the execution of the test cases.

Typically it is a matter of integration tests, because they verify the interactions between the components and therefore the application as a whole.

OSGi pre- and post-processing

In an OSGi environment the execution of integration tests is a huge challenge, because they need to be run in the OSGi framework. That's why the startup of the framework and the installation of the required bundles must be done before the test can be executed as well as a cleanup and shutdown afterwards.

Alternatives

Alternative 1: Implementation as part of the tests

A way this can be done is through the implementation of the required pre- and post processing steps as part of the test cases. Current testing frameworks provide for these purposes specific methods, which allow the encapsulation of test cases in such processing steps.

Advantage	Disadvantage
• The testing framework of the developer's choice can be used. • Specific configurations and implementations of the platforms can be considered. • Can be reused if those steps are implemented in a general manner.	• Requires a good understanding of the used platforms and libraries for their setup and teardown. • May cause huge efforts for the implementation of such a behavior. • The complexity of these steps can cause errors itself. • Dependencies must be resolved, for example from the local Maven repository.

Alternative 2: Using the Spring DM testing framework

An alternative way is using Spring DM. For this purpose it provides with its testing framework the facilities which handles the pre- and post processing for the execution of integration tests in OSGi. The required configurations must be done in the class itself through overriding specific methods.

Advantage	Disadvantage
• Provides an automation of the frequently used pre- and post processing, required for the OSGi environment. • Dependent plug-ins and bundles can be loaded directly out of the local Maven repository. • Allows an easy and flexible configuration of the setup pre- and post-processing.	• At the moment, test cases can be implemented only using JUnit. • Configurations can be made only in the test classes itself. Other ways are not supported at the moment.

5 Requirements

Overview

This chapter contains the requirements for the different parts of the reference architecture. They are divided into functional and non-functional requirements, whereas the latter ones are further categorized in the main topics defined by the ISO9126 *[ISO9126]* standard for software quality.

5.1 Requirements for the Infrastructure

Overview

This chapter contains the detailed requirements for the infrastructure of the reference architecture. This covers functional requirements as well as non-functional ones.

Functional requirements

Requirements	
Target platform	The definition of target platform for all developers must be done to avoid compatibility problems. Otherwise different versions of the OSGi framework, the Eclipse Platform and further plug-ins may be used.
SVN	For the management of the projects as well as the target platform the version control system SVN must be used.

Non functional requirements

Documentation	
Documentation	The solution of our infrastructure setup as well as a developer's guide, explaining how this can be done itself, must be documented in English. It must be understandable for developers.

Functionality	
Suitability	The target platform and the remaining infrastructure should be as compact as possible to allow a quick installation.
Compatibility	The used frameworks and libraries should be compatible with the platform versions to avoid problems.

Reliability	
Availability	The infrastructure should be available at any time, even when the version control system is down.
Maturity	The target platform should not only consist of the Eclipse platform itself, but also of additional and frequently used plug-ins and libraries, like Spring and further more.

Maintainability	
Modifiability	The target platform should be composed in a way that allows its extension with new plug-ins as well as whose replacement.

Portability	
Replaceability	The replacement of the target platform should be possible with little effort.
Reusability	Central parts of the infrastructure should be managed by Subversion to allow its use for all developers.
Installability	The target platform and the remaining parts of the infrastructure should be as compact as possible to allow an easy and compact installation on a different computer.

5.2 Requirements for the Automatic Build

Overview

This chapter contains the detailed requirements for the automatic build environment of the reference architecture. This covers functional requirements as well as non-functional ones.

Functional requirements

Requirements	
Maven	For the automatic build, Maven must be used as its basis. Otherwise the compatibility with other HERAS^{AF} projects cannot be guaranteed.
Consideration of OSGi characteristics	The Maven build should consider the OSGi characteristics to allow the creation of complete and correct bundles. For this purpose different plug-ins and further procedures should be checked.
Support product generation	The automatic build should also allow the generation of deliverable and executable products like features or whole applications.
Execution of unit- and integration tests	Like in the conventional Maven build also the execution of unit- and integration tests as part of the automatic build environment should be possible. For this purpose it should consider the OSGi-specific characteristics.

Non functional requirements

Documentation	
Documentation	The solution of our automatic build should be documented in English, explaining our decisions and considerations of different alternatives. In the developer's guide the way how this can be done itself must be documented. This all must be understandable for developers.

Functionality	
Suitability	The use of the automatic build should be compact and easy in terms of its usability and configurability.
Compatibility	The extensions of the Maven build must be compatible with the used platforms and frameworks.
Interoperability	All the used plug-ins and technologies must be invoked from or integrated in the conventional Maven build.

Reliability	
Fault Tolerance	If errors in the build process occur it should be abort. Additionally they should be printed out to allow their correction.
Maturity	The automatic build process should be stable covering all expected situations and conditions.

Usability	
Understandability	The use of the automatic build should be the same as that of the Maven build.

Maintainability	
Analysability	It must be possible to locate the source of an error easily. Therefore the build should print out the executed steps and the errors itself.

Portability	
Adaptability	The configurations of the build extensions should be made in way to reduce duplications.
Installability	The installation of the automatic build should be as compact as possible.

5.3 Requirements for the Reference Architecture

Overview

This chapter contains the detailed requirements for the reference architecture itself. This covers functional requirements as well as non-functional ones.

Functional requirements

Requirements	
Rich Client Platform	For the implementation of the reference architecture and especially the rich client, Eclipse RCP should be used.
Encapsulation of con-	The different concerns of an application should be

	cerns	encapsulated to allow an easy addition as well as their replacement. Also direct dependencies to specific classes should be avoided.
	Configuration over implementation	Instead of platform-specific implementations neutral configurations like extension points should be used as much as possible.
	Component integration	The integration as well as the removal of components into a RCP application should be possible without several adaptions.
	Use of text editor	A text editor for the entry of structured text, whose conversion into a domain model should be possible. Appropriate features, like syntax highlighting, supporting the user should be considered as well.
	Use of graphical editor	Also a graphical editor for the visualization of domain models and their manipulation should be implemented. Appropriate features, like palettes and direct edit, supporting the user should be considered as well.
	Synchronization of the editors using a common domain model	Common parts like a domain model should be accessible and managed in a central component. This includes also the synchronization of dependent components to allow updating their views.
	Tutorial applications	Tutorial applications should be developed to prove the concepts as well as the design decisions.
Non functional requirements	**Documentation**	
	Documentation	The chosen solution for the reference architecture including an architectural schema as well as the conception of application components should be documented in English. Especially the different considerations and decisions must be explained in detail as well as a developer's guide showing how this can be done itself. This all must be understandable for developers. Additionally the tutorial applications, which will be developed to prove the decisions and concepts, should be documented with Javadoc in English.
	Functionality	
	Suitability	The chosen architecture for future RCP application should be appropriate for small applications as well as for complex ones.
	Compatibility	The use of explicit versions in the dependencies should be avoided in order to allow an easier exchange of the underlying platform, plug-ins, components, etc.

Performance	
Scalability	The architecture should be chosen to support a large amount of components in the same way as a small amount. The addition of new components should therefore not lead to a worse time behavior.

Maintainability	
Analysability	The reference architecture should allow the analysis of its state through appropriate mechanisms, like logging, a clear separation of responsibilities and further more.
Testability	The architecture should be designed in a way, which allows its verification with unit and integration tests without adaptions.

Portability	
Adaptability	The architecture must allow its adaption and extension to satisfy several requirements without breaking the main concepts.
Reusability	The different components should be encapsulated to increase their reusability as well as their replacement.

5.4 Requirements for the Tests

Overview

This chapter contains the detailed requirements for the testing of the reference architecture. This covers functional requirements as well as non-functional ones.

Functional requirements

Requirements	
Unit test	It should be possible to implement unit tests for the verification of selected parts.
Integration test	Also the development of application wide integration tests considering the OSGi environment should be possible.

Non functional requirements

Documentation	
Documentation	The solution of our unit- and integration test should be documented in English. Therefore the explanations of the different considerations and decisions must be contained. In the developer's guide the way how this can be done itself must be documented. This all must be understandable for developers.

Functionality	
Suitability	
Compatibility	The integration test should be configurable to use on any OSGi environment which corresponds the official specification.
Interoperability	The integration tests should be executed in cooperation with the underlying OSGi environment.

Usability	
Understandability	The configuration of the unit and integration test should be simple and compact.
Operability	It should be possible to integrate the test invocation in the automatic build.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Part III: Reference Architecture

6 Solution

Overview

In this chapter we document in detail the development of our applications. This covers in particular the comparison of alternatives and the decision for one of them as well as occurred problems and our solutions for them. The chapter is structured by layers to group the different themes logically.

6.1 Infrastructure

Overview

This chapter describes the different topics related to the infrastructure as a part of the reference architecture.

6.1.1 Automatic Build

Intention

An important part of each development environment is the build environment. On the one hand it facilitates the daily work of the developers through the automation of steady tasks; on the other hand it allows a continuous integration with builds-on-check-in, nightly builds and furthers more.

A special challenge in this case is the combination of the bundle development with its peculiarities and the Maven tools with its dependency management. In the following chapter, we go into the different parts of the automatic build in the order listed below.

- Conventional Maven build with its dependency management
- Maven Bundle Plug-in to support the bundle handling
- Headless PDE Build for the product generation
- Automatic Integration- and Unit testing for OSGi and RCP bundles (see chapter 6.6)

Overview

The following **Figure 5** shows the schematic workflow of the developed build process in a chronological order. This includes also the product generation. Individual steps only to build the bundles for example, can skip some of the steps of the whole process.

To run the build process we required additional tools. Besides the Maven build tool and its repositories there are also the a target platform to run the tests on it and an Eclipse instance which provides the PDE build tools for the product generation.

Now we take a detailed look at the build process. Below the figure all tagged steps are explained.

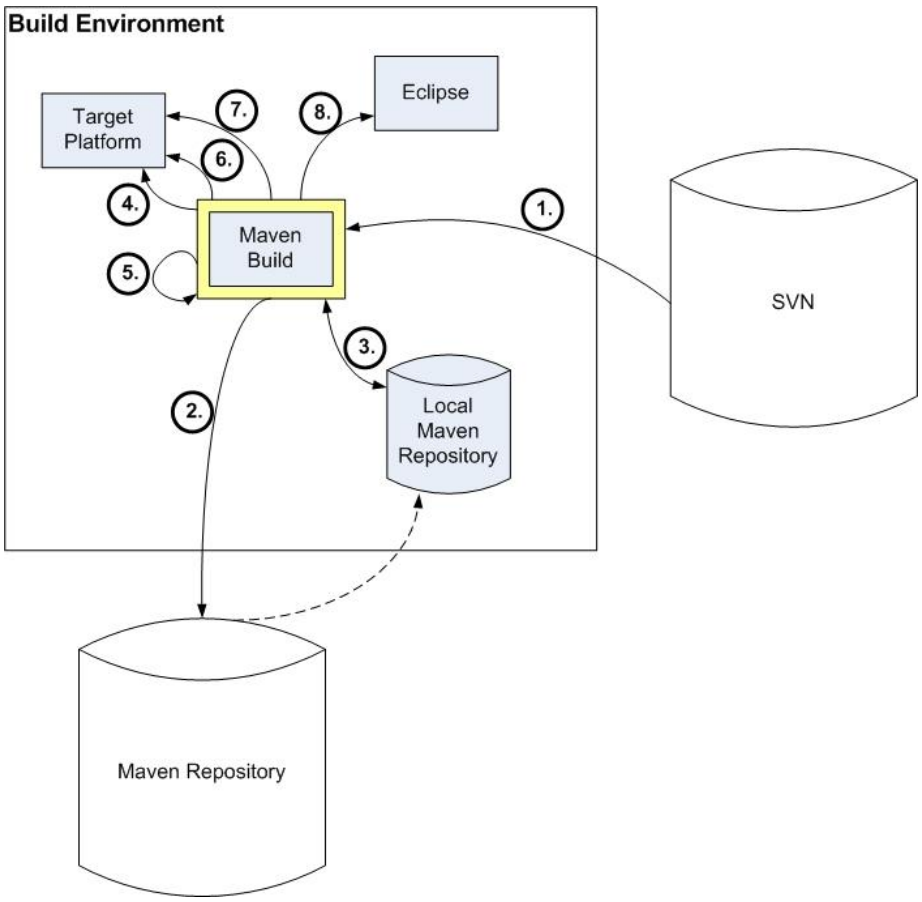

Figure 5: Schematic workflow of the automatic build environment

1. Checkout of the Plug-in Project which should be automatically built by the build environment.
2. Download the required bundles and Java archives from the remote Maven repositories into the local one.
3. Get the required bundles and Java archives from the local Maven repository.
4. Setup the target platform to run the tests against it. In our solution, the target platform is integrated in the local Maven repository. The reason for this collocation is the Spring DM testing framework. (see chapter 6.6)
5. Compile the source code of the project and build bundles with the referenced bundles and Java archives out of the local Maven repository.
6. Install the bundles in the target platform to run the integration tests.
7. Run the tests against the target platform with the built bundles for verification purposes.
8. Build executable and deliverable products using the PDE Build tools provided by Eclipse.

Except for step 8, all others are executed by Maven and the Maven Bundle Plug-in. The last step is triggered by Maven but executed by the PDE Build provided by Eclipse.

6.1.1.1 Maven

Intention

Maven is an extension of the well-known Ant Build Tool, which extends the approach from Ant with an automatic Ant script generation and an integrated dependency management. The plug-in architecture of Maven allows additionally its extension with additional plug-ins to enhance the basic functionality with specific ones, like the support for the bundle handling. This and the fact of its wide spreading were the main reasons to use Maven as the main build tool.

Repository

Therefore we used it as the basic build tool to provide the standard build steps like compiling and building JAR files as well as for the dependency management.

To use this Maven functionality, the bundles from the target platform, they are also JAR files but with an extended manifest file, have to be managed in a Maven repository. Since the Eclipse foundation doesn't provide the bundles of their Eclipse platform in a Maven repository and the public repositories contain only older versions, we had to add them in the HERAS^AF repository[4], which we used primarily.

To build the bundles, the dependent ones can be handled now like the dependencies in other Java applications. The only point to consider is the looser coupling of the dependencies in an OSGi environment. Through the definition of the import packages instead of the bundles itself, the *pom.xml* files may contain not all required bundles. This can lead to errors during the build process, which can be solved with the addition of the missing bundles to the dependencies of your own bundle for the compilation phase.

6.1.1.1.1 Structure of the build scripts

Intention

For a professional use of the Maven build tool, the structure of the build scripts has to be considered as well. A wrong or suboptimal structure leads to duplicate configurations in the scripts which increase the risk of errors. Additionally synergies between the build scripts can be used if the common parts are located in a higher layer.

Alternatives

Alternative 1: Simple *pom.xml* files

A first solution for the organization of the build scripts is the use of simple *pom.xml* files. One for each project respectively bundle/feature which contains all required plug-ins, resources and dependencies. No dependencies to other *pom.xml* files exist which may cause side effects.

Advantage	Disadvantage
• The *pom.xml* files contain all required plug-ins, resources and dependencies. So the projects can be run independent from the other ones. • No side effects through	• No synergies can be used to reduce the configuration overhead, because all projects contain all of the required plug-ins, resources and dependencies. • Changes in common parts of the

[4] The HERAS^AF repository is the Maven repository of the HERAS^AF project. More details about HERAS^AF are available under: http://www.herasaf.org/

changes in other *pom.xml* files occur, cause of the independency of each project.	*pom.xml* configuration have to be made several times, because they are not concentrated in a single file. • An integrated build, consisting of all projects of an application cannot be done, because they have to be run separately. Additionally the definition of a build sequence is hard to achieve.

Alternative 2: Inheritance hierarchy between the *pom.xml*

A second solution is the development of an inheritance hierarchy between the *pom.xml* files of the projects. The definition of parent projects allows the location of common parts in them and allows all derived projects to use them. The use of project global dependencies is restricted to those of the inheritance between the projects.

Advantage	Disadvantage
• Through the use of parent projects, common build configurations have to be made only once but can be used several times. • Through the inheritance, the configuration overhead can be reduced to a minimum. This means each *pom.xml* can focus on its key concerns. • The use of an integrated build is possible through the build of the parent projects. This allows a continuous integration and the exact control of the build sequence. • The versions of the used bundles can be management easily as properties in the *pom.xml* of the parent project. This allows an easier migration to another version.	• If the hierarchy of the *pom.xml* files is handled with Eclipse projects, additional projects are used, which contain them. • The *pom.xml* files of the projects are not independent anymore.

Decision

Based on the comparisons above, we decided to split our scripts to build at least two layers. The parent script in the highest layer contains the jointly used properties as well as dependencies and defines the plug-ins to be used in the derived scripts of each project.

Additionally, we can locate the repository declarations in the parent scripts too, to allow all derived scripts to use them if they need them. Contrary to the parent scripts, the ones in the derived projects can focus on its specific configurations. They contain only the specific dependencies as well as special plug-ins and repositories. If needed they override additionally the properties of the parent

script with its individual values.

In a solution without the inheritance hierarchy, these configurations have to be made several times in each pom file, which causes unnecessary duplicity. Changes of the configuration have to be made several times too. These and the additional disadvantages of the independent solution promoted us to use the inheritance solution.

6.1.1.1.2 Problems

Class path conflicts

Special problems which appeared were the classpath conflicts in the Eclipse platform during the implementation of the bundles. The causes for these problems were the missing compatibility of the Eclipse Maven Plug-in with the rest of the Plug-in development in Eclipse. This leads to duplicate entries in the M*aven Classpath Container* and the *Plug-in Dependencies* which cannot be handled by Eclipse. The removal of the *Maven Classpath Container* solved this problem but leads to a further problem.

Unit tests

Through the remove of the *Maven Classpath Container*, it was no longer possible to implement the unit tests in the bundles itself. To do this it would be necessary to add the testing bundles to the *Plug-in Dependencies*. But such a solution is inappropriate and unusable. That's why we decided to locate the unit tests besides the integration tests also in own projects. In the future it might be possible to locate the unit tests in the bundles itself after an improved coordination between the involved plug-ins. But at the moment it's not yet possible.

Versions of the used bundles and plug-ins

Another problem occurred during the development of the Automatic Build, was the management of the bundle version numbers. In cases of a migration to a newer version of the target platform, the version numbers have to be changed in each *pom.xml* file. In a small application, this is no problem, but in bigger ones, this can lead to an increased effort.

We could solve this problem through the definition of properties in the *pom.xml* of the parent project, containing the concrete version numbers. In the derived *pom.xml* files, only the properties should be used. To change now the versions of the used bundles, only the properties have to be adjusted.

6.1.1.2 Maven Bundle Plug-in

Intention

Maven supports the developer of Java applications with its dependency management and other features. A similar support for the development of OSGi bundles and RCP plug-ins is also desirable. For this reason, several projects provide supporting tools like Maven plug-ins to facilitate the development of OSGi bundles and RCP plug-ins.

In this chapter, we take a look at the different Maven plug-ins, show you which ones we decided to use and how we used them. Additionally we describe the occurred problems.

6.1.1.2.1 Bundle Plug-ins

Overview

The integration of the bundle development in the existing build process requires its adaption. Otherwise it is not able to consider the specific characteristics of bundles, fragments, features, and further more. The decision which integration tool to use, has a direct impact on the existing build process. An incompatible or inappropriate integration tool can cause more disadvantages than advantages.

Therefore an important requirement for the integration tool to use is the seamless integration in the existing Maven build process. Otherwise it would be hard to configure and use it.

In the following comparison, we balance different integration tools and plug-ins against each other in detail. After a short recap we describe our decision about the integration tool we plan to use.

Alternatives

Alternative 1: Without a plug-in

A first and obvious solution is to develop OSGi bundles and RCP plug-ins without the support of an integration tool. The abdication of an integration tool brings simplifications as well as consequences. An important consequence is the missing consideration of the specific bundle characteristics during the build, to mention only one.

It requires a manual configuration of the bundles and its dependencies and doesn't support the developer during the development of OSGi bundles and RCP plug-ins.

Advantage	Disadvantage
• No special Maven configuration is needed. To build the application, the existing Maven plug-ins must be used. • During the implementation, only the manifest has to be adjusted with the required imports and exports.	• The existing Maven plug-ins may be inappropriate, because they don't consider the special requirements of the OSGi bundles and RCP plug-ins. This can lead to additional manually configurations. • Automatic builds may be hard to achieve, especially features which can only be tested in an OSGi environment.

Alternative 2: Apache Felix - Bundle Plug-in for Maven

The Apache implementation of the OSGi specification under the name Felix provides a special Maven plug-in, the Bundle Plugin for Maven *[MVNBundle]*.

The plug-in doesn't depend on the specific Apache Felix implementation. Therefore it can be used also in combination with the Equinox, Knopflerfish and further OSGi implementations. The use of the plug-in is not different to others.

Advantage	Disadvantage
• Generates the Manifest file with the contents declared in the *pom.xml*. • The plug-in provides the	• The dependencies are not automatically added to the manifest. The reason is that the plug-in doesn't know which kind OSGi dependency (*Import-Package*, *Require-Bundle*,

packaging type bundle, which creates bundles instead of normal Java archive files. • Declared resources are automatically added to the bundles. • The plug-in is well integrated in Maven and its build process.	etc.) it should add in the manifest for a given Maven dependency. • The *Maven Classpath Container* has to be removed, otherwise conflicts in Eclipse occur.

Alternative 3: ETHZ Maven-OSGi Plug-in

Another Maven plug-in supporting the development of OSGi bundles and Eclipse plug-ins is the Maven-OSGi Plugin *[MVNOSGi]* provided the ETH Zurich. It was one of the first Maven plug-ins which supported the developers of bundles and plug-ins. Its main and also only purpose is the generation of the manifest file for bundles and plug-ins. It doesn't support the developer with further integration steps.

Advantage	Disadvantage
• It provides a hierarchical definition of the properties for the OSGi manifest. • It allows the integration of Maven dependencies in the *Bundle-Classpath* by declaration.	• The plug-in breaks the XML schema definition of the pom through the addition of several tags. • The plug-in was implemented in 2005 and since then not enhanced or supported. • The plug-in only provides the manifest generation. Further integration steps like the creation of bundle jars are not supported. • The plug-in was implemented against an old Maven version. It is likely that it doesn't run with the current version.

Alternative 4: Tycho Plug-in

Also a plug-in supporting the development of OSGi bundles and Eclipse plug-ins is Tycho *[Tycho]*. The plug-in brings an extensive integration of several development steps. This covers the manifest generation, the export as bundles, features, fragments, Eclipse features, Eclipse update sites and Eclipse products (RCP applications). At the moment the plug-in is under construction and provides only a small amount of these functions.

Advantage	Disadvantage
• The plug-in brings the most extensive integration of all plug-ins including the export of bundles, features, fragments, and further more.	• The plug-in is still under construction and therefore provides only a few of the described functions. • It is still not clear whether the plug-in will provide all the announced

- The plug-in brings several new packaging types for the different products to create.
- It provides a manifest-first and a pom-first development.

features.

Decision

Based on the comparison above, the manual implementation is not a manner of developing bundles and plug-ins as well as the Maven-OSGi Plugin provided by the ETH Zurich. They both do not provide real integration steps for the build process.

From the remaining plug-ins, *Tycho* would be a very attractive solution, because it provides the most extensive integration with several automation steps. The most of the provided steps have to be made manually or step by step when using other integration tools. Especially the automatic build of bundles, features, fragments and the several Eclipse products is a revolutionary change. Unfortunately the plug-in is still under construction and provides at the moment only a small amount of the announced features and functions.

Other the *Bundle Plugin for Maven* provided by the Apache Felix implementation. It only provides the manifest generation and the build of several OSGi products like bundles and fragments. The build of large-scale products like features and Eclipse products is not available and has to be solved in other ways, especially with the PDE Build.

Based on these facts, we decided to use the *Bundle Plugin for Maven*, because it provides the best solution at the time this thesis was written. The plug-in is stable and well used in the developer community. But in the future, its replacement with the Tycho plug-in should be considered, if this can provide all or most of the announced features and functions.

6.1.1.2.2 Use of the Bundle Plug-in

Overview

Like mentioned in the chapters above, we decided to split the pom files in a hierarchical structure. This allows us to minimize duplicate configurations. Additionally we decided to use the *Bundle Plugin for Maven* provided by the Apache Felix project. Although the features and functionality of the plug-in are limited it provides a great simplification of the daily development and is the best available plug-in at the moment.

The following **Figure 6** shows the project structure schematically. The important changes which are accompanied with the integration of the Bundle plug-in are written in the project folders.

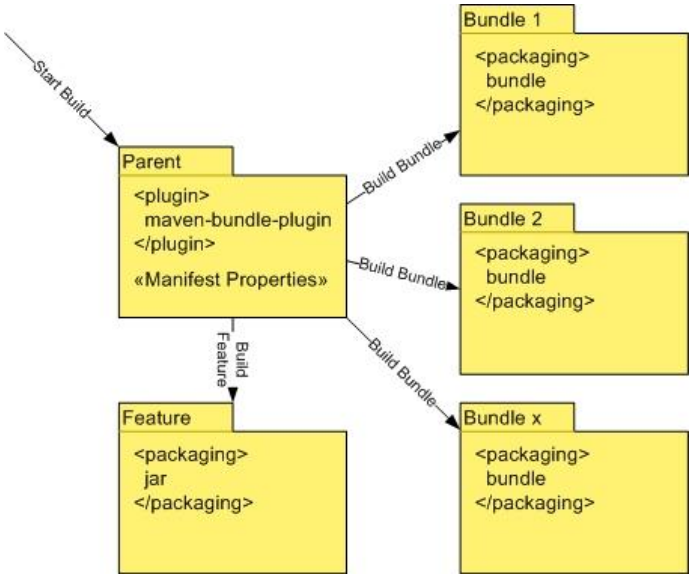

Figure 6: Bundle plug-in integration in the build process

Plug-in

The use of the Maven plug-in is now very easy. Maven downloads and installs the plug-in itself if it is listed in the `<plugins>` tag.

Following the decision of the hierarchical use (see chapter 6.1.1.1.1) of the pom files, we placed the plug-in in the parent file. The definition of the plug-in in the parent doesn't lead to problems, because it will only be executed in cases where the derived pom files use the packaging type *bundle*.

```
<packaging>bundle</packaging>
```

Additionally to the conventional plug-in tags, this one requires additional configuration instructions in the tag `<instructions>`. The contained tags are added as entries in the generated manifest file.

The names of the tags are the names of the properties in the manifest file and the values are consequently the values in the manifest file. Additionally the tags are not defined in the XML schema of Maven or the plug-in, which allows a generic and flexible use of them. If a new property in the manifest file is used, it can be added simply as new tag.

A detailed description of the specified OSGi properties and how their content has to be configured is listed in chapter 8.8.3.1.

To allow the definition and configuration of the plug-in in the manifest file, we had to define properties in the same file. They act as placeholder for the concrete values which are set in the derived pom files. The following snippet shows an example configuration.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>2.0.0</version>
      <extensions>true</extensions>
      <configuration>
        <manifestLocation>META-INF</manifestLocation>
```

```
        <instructions>
          <Bundle-Name>
            ${artifactId}
          </Bundle-Name>
          <Bundle-SymbolicName>
            ${manifest.symbolicname}
          </Bundle-SymbolicName>
          <Import-Package>
            ${manifest.import}
          </Import-Package>
          <Export-Package>
            ${manifest.export}
          </Export-Package>
          <Private-Package>
            ${manifest.private}
          </Private-Package>
          <Bundle-Activator>
            ${manifest.activator}
          </Bundle-Activator>
          <Require-Bundle>
            ${manifest.require}
          </Require-Bundle>
          <Include-Resource>
            ${manifest.resource}
          </Include-Resource>
          <Bundle-RequiredExecutionEnvironment>
            ${manifest.environment}
          </Bundle-RequiredExecutionEnvironment>
          <Bundle-ActivationPolicy>
            ${manifest.activationPolicy}
          </Bundle-ActivationPolicy>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Listing 1: Snippet of the Bundle Plugin for Maven in the pom.xml

Properties

The properties used in the plug-in configuration above have to be defined in advance. Otherwise they cannot be used in the instructions tag.

We added the definition in the same pom file we also defined the *Bundle Plugin for Maven*. This gave us the additional chance to define common values for all bundles. In the derived pom files we only had to override the properties we are really using or whose values are different from the ones in the parent. The others are taken from the parent definition.

The following snippet shows an example of the properties definition in the parent pom file.

```
<properties>
  <manifest.symbolicname>
    ${project.groupId}.${project.artifactId}
  </manifest.symbolicname>
  <manifest.private></manifest.private>
  <manifest.export></manifest.export>
  <manifest.import>
    !${manifest.export},
    *;resolution:=optional
  </manifest.import>
  <manifest.activator></manifest.activator>
  <manifest.require></manifest.require>
```

```
  <manifest.resource></manifest.resource>
  <manifest.classpath>.</manifest.classpath>
  <manifest.environment>J2SE-1.5</manifest.environment>
  <manifest.activationPolicy></manifest.activationPolicy>

  <!-- Other properties -->
  ...
</properties>
```

Listing 2: Snippet of the properties definition in the parent pom.xml

6.1.1.2.3 Problems

During the configuration and use of the *Bundle Plugin for Maven*, we had several problems to solve.

Inheritance

One of the biggest of them was the problem with the **inheritance** of the pom files. The default version 1.4.3 of the plug-in was not able to handle an inheritance structure of the pom files. The new version 2.0.0 which was published at the end of February 2009 supports now an inheritance structure.

Manifest generation

Another curious problem occurred during the **manifest generation**. Some of the changed properties in the pom file were not included in the manifest. A repeated generation of the manifest couldn't solve the problem either. Only the delete of the existing manifest in the project folder and the subsequently creation of a new one was successful.

Import-Package
Export-Package

Also problems caused the definition of the *Import-Package* and *Export-Package* definitions in the pom file. In some cases not described packages were added to one of these manifest properties, the *Private-Package* or the *Ignore-Package*.

Also, the definition of the *Import-Package* and *Export-Package* headers in the pom file caused problems. In some cases not specified packages were added either to the *Private-Package* or to the *Ignore-Package* header.

We could solve the wrong assignation of the packages in the manifest by a clear definition of them in the properties of the pom. If there are packages which are not considered in the pom properties, the Bundle plug-in makes an automatic assignation in the manifest.

Split packages

A huge problem which occurred during the implementation of bundles was the one with the **split packages**. The problem occurs if two or more bundles contain classes in the same packages and export them. Another bundle importing this package cannot decide from which bundle the package should be imported. In this case the *Bundle Plugin for Maven* decides for itself which one it wants to take and adds in the manifest the following restriction to the import package.

```
Bundle-Activator: org.herasaf.tutorial.calculator.ui.Activator
Import-Package: org.eclipse.core.commands.common,
 org.eclipse.core.runtime;registry=split;version="3.4",
 org.eclipse.equinox.app;version="1.0",
 org.eclipse.jface.action,
 ...
```

Listing 3: Snippet of a generated manifest with split packages

In this case only the listed import package from the bundle `org.eclipse.equinox.registry` will be used. This may lead to problems if classes out of other bundles are used. This problem occurs especially when using classes out of the package `org.eclipse.core.runtime`. Classes of it are shared among the following bundles.

- org.eclipse.core.runtime
- org.eclipse.equinox.common
- org.eclipse.equinox.registry

A list of further packages distributed among different plug-ins provides the Eclipse Help in the Platform Plug-in Developer Guide *[PluginDevGuide]*.

According to different websites, the problem can be solved in different ways. We tried them and give you here the result of our experiences. All in all only the last solution worked well.

- The first recommendation was to add the split directive manually to the package name in the pom properties, as shown below. This solution doesn't work, because only the first `<bundle>=split` directive will be considered through the Eclipse and the OSGi environment.

```
<manifest.import>
  ...
  org.eclipse.core.commands.common,
  org.eclipse.core.runtime;common=split;registry=split;
    runtime=split;version="3.4",
  org.eclipse.equinox.app,
  ...
</manifest.import>
```

- The second advice consisted of the addition of the bundles which provide classes in the same packages as required bundles. This solution didn't work very well, because the *Bundle Plug-in for Maven* adds the package automatically to the *Import-Package* directive. In this case the package would be imported through the import and the bundle directive. The problem occurring in this case is described in the OSGi Service Platform Core Specification *[OSGiCore]* as follows: *"A bundle may both import packages (via Import-Package) and require one or more bundles (via Require-Bundle), but if a package is imported via Import-Package, it is not also visible via Require-Bundle: Import-Package takes priority over Require-Bundle, and packages which are exported by a required bundle and imported via Import-Package must not be treated as split packages."*

```
<manifest.require>
  org.eclipse.ui.workbench,
  org.eclipse.core.runtime;bundle-version="3.4.0",
  org.eclipse.equinox.common,
  org.eclipse.equinox.registry
</manifest.require>
```

- The last recommendation to solve this problem was to add the bundles providing the different parts of the split packages in the *Require-Bundle* directive and to use in the *Import-Package* a negative pattern for the

package affected by the split package problem *[Schor08]*. In this case the package won't be added in the *Import-Package* directive and therefore the *Require-Bundle* will come to use. It is only a workaround but after some trials it worked very well.
Unfortunately it increases the dependency between the bundles.

```
<manifest.import>
  ...,
  !org.eclipse.core.runtime;version="3.4.0",
  org.eclipse.equinox.app,
  ...
</manifest.import>

<manifest.require>
  org.eclipse.core.runtime;bundle-version="3.4.0",
  ...
</manifest.require>
```

Manifest formatting

Another very strange problem occurred during the use of the *Bundle Plug-in for Maven*, were inexplicable errors in Eclipse. Although all the entries and directives in the file are correct, Eclipse cannot read them correctly. The cause of the problem is the formatting of the Manifest file, because we formatted it with the keyboard shortcut *CTRL+SHIFT+F*. This solved the problem.

6.1.1.3 Headless PDE Build

Intention

The last part of the automatic build environment includes the PDE build. Its purpose is the creation of an executable product. It's an integral part of an automatic build environment to provide a product generation, especially if they should be published regularly, like e.g. in nightly builds.

In this chapter, we take a look at the different integration tools to create executable products. We also show you which one we decided to use and how we used it. Additionally we describe the occurred problems.

6.1.1.3.1 PDE Build plug-ins and procedures

Overview

The PDE build is an Eclipse plug-in, which allows the creation of executable products as an archive in the form of the Eclipse distribution. The archive contains an executable as well as the required features and bundles. Eclipse provides an export function in the IDE to create products based on a *.product* file. Since the plug-in is available as bundle, it can be also used from outside Eclipse.

Today several integration tools like Maven plug-ins and others exist, which allows the use of the PDE Build provided by Eclipse. The decision which tool to use has impact on the way how the Headless PDE Build can be integrated in the existing build process. Therefore we balance them against each other in the following comparison.

Alternatives

Alternative 1: PDE Build using an ANT file

The first and most basic way to use the PDE build is with a simple Ant file. It consists of the setup of a build directory and the start of the Eclipse bundle,

which creates the product. It is a well-known solution and provides the full range of configurations. Unfortunately it is a basic implementation and its integration in the maven build process is more difficult to achieve than with a Maven plug-in.

The implementation of the Ant file for the product generation is described amongst others in *[Vogel09PDE]*, *[Barchfeld05]* and *[Panier07]*. It is based on the build script itself and a properties definition.

Advantage	Disadvantage
• The PDE build provided by Eclipse is based on Ant files. The use of Ant for running it is obvious. • The use of a simple Ant script to run the PDE build allows a fine grained configuration. • The build can easily be extended with preparation and post-processing steps. • The properties for controlling the PDE build can be located at several points, because the handover of them has to be made by hand.	• The integration of the Ant script in the existing build process is more difficult than an ordinary Maven plug-in. • Compared with a conventional Maven plug-in, additional scripts, which are possibly located in own projects, are required.

Alternative 2: Eclipse PDE Maven Plugin

A full integration of the PDE build in the existing build process provides the Eclipse PDE Maven Plugin *[MVNPDE]*. It's one of the first plug-ins, which integrates the product generation in the Maven build process and is well-known.

Advantage	Disadvantage
• It allows seamless integration in the existing build process through the use of an additional plug-in.	• The plug-in has very strict requirements for the project structure, which should be built. This leads to complications with SVN. • The plug-in allows only a limited possibility to configure the PDE build. Therefore the properties for controlling the PDE build can hardly be located at several points. • Although the plug-in has several errors and restrictions, it will not be maintained and enhanced anymore. • Based on several forum entries, the plug-in is very fragile. • It uses a *startup.jar* in the root directory of the eclipse. In the new version of Eclipse, this jar was moved in the plugin folder. A manual copy

has to be made.

Alternative 3: Tycho Plug-in

Also a plug-in supporting the creation of executable products is Tycho *[Tycho]*. The plug-in brings an extensive integration of several development steps in the Maven build process. This covers amongst others the creation of bundles, fragments, features, update sites and RCP applications. At the moment the plug-in is under construction and provides only a small amount of the announced features. The ones supporting the Headless PDE Build are not included.

Advantage	Disadvantage
• It allows seamless integration in the existing build process through the use of an additional plug-in. • The plug-in brings the most extensive integration for the product export. • It can not only create RCP applications as products, but also bundles, features, fragments and update sites. • The configuration of the PDE build is much easier than with the Eclipse PDE Maven Plugin.	• The plug-in is still under construction and therefore provides only a few of the described features. • It is still not clear whether the plug-in will provide all the announced features.

Decision

Based on the comparison above, the Eclipse PDE Maven Plugin can be excluded from the possible solutions. The reason is its fragility as well as the restrictions which limits the configurability and the execution of the PDE Build too much. Additionally its integration in the used infrastructure, not only the build process, is hard to achieve, because it requires a specific nested folder structure. But this conflicts with SVN to mention only tool.

The Tycho plug-in seems to be best solution for the product generation. It provides a seamless integration in the existing build process and supports not only the RCP application generation but also bundles, fragments, features and update sites. Because of that it offers the widest range of features. Unfortunately it's under construction and doesn't yet provide the announced functionality.

The development of Ant scripts allows a fine grained configuration of the PDE build. If necessary additionally a pre- and post-processing can be included easily in the build process. Although the implementation of Ant scripts brings disadvantages such as own projects and additional scripts to write, it seems the most appropriate solution at the moment, because it's approved and stable.

Based on these facts, we decided to use the Ant script to integrate the PDE build in our automatic build environment, because it provides the best solution at the moment. It can easily be configured and extended if required and its integration is through the AntRun plug-in possible. It's a stable and well-known solution in the developer community and still often supported and used. But in the future, its replacement with the Tycho plug-in should be considered, if this can provide

the announced features and functions for the PDE build integration.

6.1.1.3.2 Use of the PDE Build

Overview

Based on the decision described in the chapter above, we used a simple Ant script to run the PDE build. It allowed us a more detailed configuration of the PDE build than other tools could provide. Additionally we could define the structure of the PDE build on our own and don't have to consider restrictions of the plug-ins.

Dependency to Eclipse

To run the PDE build, the plug-in `org.eclipse.pde.build` is required in your local Eclipse installation. It is a part of the *Eclipse Plug-in Development Environment* and can also be used form outside the Eclipse IDE, through the start of a simple OSGi application.

This leads to a limitation of the build process. It can only be run on a platform where an Eclipse instance with the Plug-in Development Environment is installed.

Structure

To integrate the PDE build in the existing build process, we decided to extend the existing structure with an additional project containing the build scripts. The following **Figure 7** shows the project structure schematically.

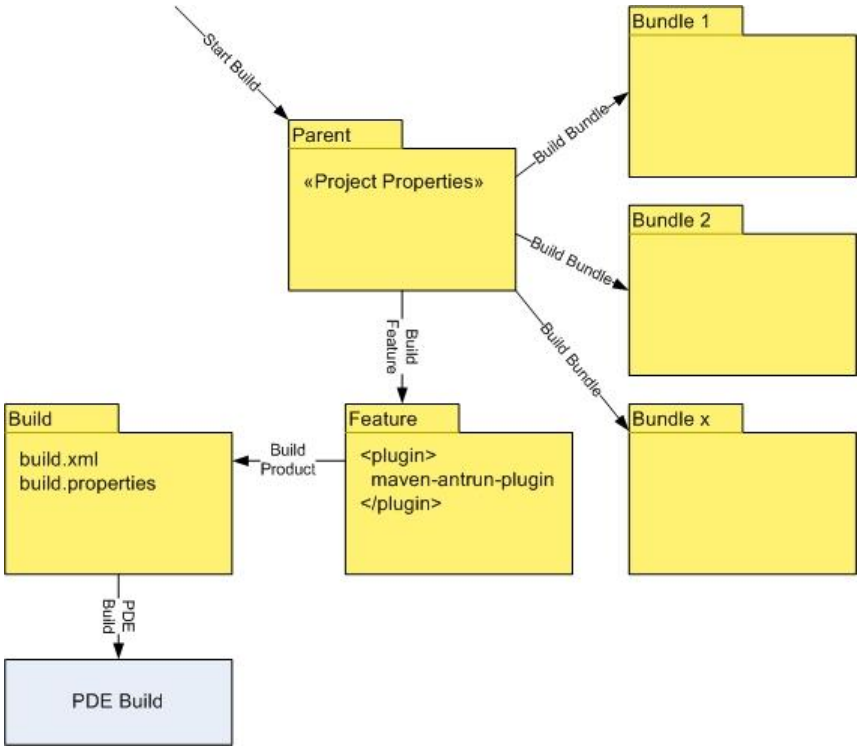

Figure 7: PDE Build integration in the build process

The parent project controls the build process insofar it configures the build sequence. It references the remaining projects (bundles, fragments, features, etc.)

of the application and hands over the goals for the Maven builds.

Additionally we created a new *Build* project. The main purpose for this new project is the definition of a neutral Headless PDE Build, which can be integrated in every project requiring a PDE build. It contains the Ant script for the PDE Build and a properties file, which defines general properties applying for all kinds of PDE Builds. The settings for the build of a concrete product, like the location of the product file, the build directory and further more are outsourced to the calling projects, which configure and pass them as properties.

To build now a concrete application with the Headless PDE Build, we created an additional feature project, which contains the Maven AntRun Plug-in. It allows the extension of the existing pom file with Ant tasks, like the invocation of Ant scripts. In this section, the feature defines the project-specific properties and calls the build script of the Headless PDE Build in the Build project.

We decided to locate the invocation of the Headless PDE Build in the feature, because an invocation from the parent project would lead to a wrong build sequence. This means the Maven build would execute the Headless PDE Build as the first step before it executes the remaining projects.

Build script

The build script itself, like mentioned above, contains no project specific properties and settings. All these things are configured outside. Therefore it is possible to write the build script once and use it several times in different projects.

Although the script is implemented generally, it has one restriction. To allow the distinction between bundles and features, the feature projects must have “feature” in their project name. Otherwise the script is not able to differentiate between the projects, because features don’t have to be compiled and packaged like bundles and fragments.

Properties

The properties in the PDE build are used to configure and control it. They define directories where the projects are, where the build directory is, where the *.product* file is and further more.

To define the Headless PDE Build free of project specific properties, we outsourced them. Only the general ones applying for all PDE builds remained in the *build.properties* of the build project. The other properties are passed from the Maven *pom.xml* to the build script, like shown in **Listing 4** below.

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <!-- Execute the product build -->
    <execution>
      <id>build-product</id>
      <phase>install</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <!-- Global settings -->
          <property name="buildDirectory"
                    value="${buildDirectory}" />
          <property name="eclipseLocation"
                    value="${eclipseLocation}" />
          <property name="equinoxLauncherPluginVersion"
                    value="${equinoxLauncherPluginVersion}" />
```

```
          <property name="pdeBuildPluginVersion"
                    value="${pdeBuildPluginVersion}" />
          <!-- Project settings -->
          <property name="topLevelElementType"
                    value="feature" />
          <property name="topLevelElementId"
                    value="Calculator" />
          <property name="product"
                    value=".../calculator.product" />
          <property name="archivePrefix"
                    value="calculator" />
          <property name="configs"
                    value="win32, win32, x86" />
          <property name="buildType"
                    value="I" />
          <property name="buildId"
                    value="Calculator" />
          <property name="base"
                    value="${targetPlatformsDirectory}" />
          <property name="baseLocation"
                    value="${base}/equinox-3.4.2" />
          <property name="baseos"
                    value="win32" />
          <property name="basews"
                    value="win32" />
          <property name="basearch"
                    value="x86" />
          <property name="javacSource"
                    value="1.5" />
          <property name="javacTarget"
                    value="1.5" />
          <property name="projectPrefix"
                    value="org.herasaf.tutorial.calculator" />

          <ant dir="$.../org.herasaf.pde.pdebuild"
               target="build" inheritRefs="true" />
        </tasks>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 4: Snippet of the properties handover from the Maven pom.xml to the Ant build script

Project specific properties are located in the pom file of the calling project. It defines them before he calls the Headless PDE Build.

The remaining properties are infrastructure specific, which means they depend on the programs and tools installed on your computer. They are defined in the *settings.xml* of your Maven installation, like shown in the snippet below.

```
  <profiles>
    ...
    <profile>
      <id>headlessPDEBuild</id>
      <properties>
        <buildDirectory>
          C:/temp
        </buildDirectory>
        <eclipseLocation>
          C:/Programme/eclipse
        </eclipseLocation>
        <!-- Version number for org.eclipse.equinox.launcher -->
```

```
        <equinoxLauncherPluginVersion>
          1.0.101.R34x_v20081125
        </equinoxLauncherPluginVersion>
        <!-- Version number for org.eclipse.pde.build -->
        <pdeBuildPluginVersion>
          3.4.1.R34x_v20081217
        </pdeBuildPluginVersion>
        <targetPlatformsDirectory>
          ${basedir}/../TargetPlatforms
        </targetPlatformsDirectory>
      </properties>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>headlessPDEBuild</activeProfile>
    ...
  </activeProfiles>
```

Listing 5: Snippet of the infrastructure properties in the settings.xml

6.1.1.3.3 Problems

During the implementation, configuration and use of the Headless PDE Build several problems occurred which had to be solved by us. Here's a listing of them.

PDE Plug-in

There were recurring problems during the use of the Headless PDE Build. These caused errors and aborted the build process. The reasons for the errors in the most cases were missing bundles and other problems with the dependencies.

They occurred because the *.product* file was not up to date. A synchronization of the product and the check of the dependencies solved the problems. Additionally the delete of the run configuration from time to time was helpful.

Handover of the Properties

Another problem was the handover of the properties from the Ant script to the Eclipse plug-in for the PDE build. The implementation of the build contains sections, where the *build.properties* file in the working directory of the calling routine will be imported.

After our outsourcing of the different properties outside this file, the PDE build was not able to read them anymore and aborted the build even if they were passed to our Ant script. Since we start the PDE build in a new process, the simple handover is not enough. So we had to pass the properties from the Ant build script to the PDE by hand, as a replacement for the file. The following example shows a snippet of it.

```
<java classname="org.eclipse.equinox.launcher.Main" fork="true"
failonerror="true">
  <arg value="-application" />
  <arg value="org.eclipse.ant.core.antRunner" />
  <arg value="-buildfile" />
  ...
  <arg value="-DbuildDirectory=${buildDirectory}" />
  <arg value="-DeclipseLocation=${eclipseLocation}" />
  ...
```

After the manual transfer of the infrastructure- and project-based properties, which are defined outside the *build.properties* of the Headless PDE Build, the problem could be solved.

Cycles between bundles

Because of cycles in external bundles, we were using, several times errors occured. By default, the PDE build is not able handle cycles between bundles and aborts the further build.

Fortunately this behavior can be configured via a property called *allowBinaryCycle*. After setting it to true the PDE build doesn't consider cycles in external bundles anymore.

```
<arg value="-DallowBinaryCycle=true" />
```

Split packages

Also problems caused the split packages in the manifest, because some of the classes were not available in the build class path. It is the same matter of problem described in chapter 6.1.1.2.3. Through the use of classes out of a package, which is distributed in more than one bundle, the *Bundle Plug-in for Maven* decides for itself which one he wants to take and adds in the manifest the restriction `...=split`.

The only solution which solved this problem was the addition of the bundles, which provide the same package, as *Require-Bundle* and a negative declaration of the package name in the *Import-Package* instruction. The other solutions described in chapter 6.1.1.2.3 don't work very well.

config.ini

A confusing problem which occurred after the Headless PDE Build, affected by the *config.ini*. It defines which bundles should be started, when the application starts.

When we started our built product, errors and exceptions occurred, because some of the bundles and plug-ins were apparently not available, even if they were in the *plugins* folder. The cause of this problem was an incomplete *config.ini* file, which doesn't define the appropriate bundles to run at startup.

To solve this problem we had to do two different steps.

1. We had to define an own *config.ini* file in the branding plug-in project, which orders the required bundles to run at startup. To avoid the inclusion of all existing bundles manually, the plug-in `org.eclipse.update.configurator` can be included with a higher start level in the startup directive of the *config.ini*. This plug-in recognizes all bundles in the *plugins* folder and publishes them to the executing application. For further information see *[Daum2008]*.
2. We checked and updated the dependant plug-ins and fragments in the *.product* file, because an incomplete list of them can lead to an incomplete *config.ini*.

Missing plug-in

Further problems which occurred during the PDE build were errors caused by the missing `org.eclipse.core.filesystem` plug-in. Even if we installed the Delta Package in our target platform, the plug-in was neither contained in Equinox, in the Eclipse Platform (RCP) nor in the Delta Package. So we had to copy it from our installed Eclipse instance in the target platform.

Headless PDE

We tried to build the Headless PDE Build as a Java archive file and to use it as

Build in a JAR

a dependency in the AntRun Plug-in to replace the direct project dependency to the Headless PDE Build project. The build itself worked fine as well as the addition to the plug-in dependencies.

Unfortunately Ant is not able to invoke Ant files which are contained in archives. To execute the Headless PDE Build from the built JAR file, it would be necessary to extract its content and to invoke afterwards the Ant script. But this would lead to huge and fragile implementation of the Headless PDE Build invocation. So we decided to omit this feature.

Maven managed feature project

During the implementation of the Headless PDE Build with the corresponding feature project, we had the problem we could not convert a Maven project to a feature. Other than the conversion to a plug-in project, the Maven plug-in provides no support for that.

To solve this problem we had to create the feature as a Feature Project in Eclipse and to complement the *.project* file with the snippets shown in **Listing 6**.

```
<projectDescription>
  <name>org.herasaf.tutorial.calculator.feature</name>
  ...
  <buildSpec>
    ...
    <buildCommand>
      <name>
        org.eclipse.iam.jdt.core.mavenIncrementalBuilder
      </name>
      <arguments>
      </arguments>
    </buildCommand>
  </buildSpec>
  <natures>
    ...
    <nature>
      org.eclipse.iam.jdt.core.mavenNature
    </nature>
  </natures>
</projectDescription>
```

Listing 6: Snippet of the .project file to with the Maven configurations of the project

After a refresh of the feature project, it was managed by Maven and could be used so.

Manifest formatting

Unfortunately the manifest formatting problem in Eclipse already described in chapter 6.1.1.2.3 has direct impact on the PDE Build itself. It seems the PDE Build uses the same mechanism to read the Manifest file, like the Eclipse IDE. For this reason our Headless PDE Build failed in cases where previously a manifest is generated, which cannot be interpreted by Eclipse.

We tried to solve this problem, but this was not possible. Also an extensive research was not successful. Although we found forum entries from users which had the same problem, there was nowhere a solution for this problem.

So we had to invoke the project build using the *Bundle Plug-in for Maven* separately from the Headless PDE Build.

6.2 Architecture

Overview

The architecture is an important part of the application design. It has a direct impact on its adaptability, maintainability and stability through the clear separation of concerns.

But there is no general solution for an appropriate architecture. Its structure depends mainly on the concrete requirements for a specific application and should be considered when they are designed.

A wrong or inappropriate architecture of an application can lead to several problems with multiple consequences. This begins with an *increased coupling* between the classes and lead to *cyclic dependencies* or at least *missing exchangeability of the layers*.

6.2.1 Layers

Intention

We also had to do the considerations mentioned in the introduction of this chapter for our tutorial applications. Although the negative impact through an inappropriate architecture was limited, because of the lower size of our tutorials, a deliberate choice helped us in a substantial manner.

Structure

The main structure we had chosen for our tutorials consisted of a separation of the UI layer with its dependencies to RCP, the control layer with specific behavior for the UI and the service layer providing and implementing OSGi based services. The **Figure 8** shows the chosen structure in a more detailed manner.

The reason for this separation is the concentration of the different layers on its core concerns.

- The **UI layer** consists mainly of the implementation of the view defined by the MVC Pattern *[MVC07]*. The classes implement the different user interfaces based on the used technologies. In our cases this is the Rich Client Platform provided by Eclipse.
- The **control layer** is a mediator between the UI and the service layer. Referred to the MVC Pattern it covers the controller and model. The classes implement the user interface specific business logic and behavior as well as a local data storage if such is needed.
- The last part is the **service layer**. It consists of the service declaration and its implementation, managed by the OSGi environment. The concrete implementation of the services depends on its requirements.

Another layering of the different components of an application is possible but should also consider a clear separation of its concerns. Otherwise an easy extension or the exchange of whole layers with different implementations, especially the view, can only be achieved with extensive changes in the existing code base.

Layers

The **Figure 8** below shows the detailed layering of the chosen architecture. For an easier wiring of the classes in and between the layers we used the Spring Framework with its Spring Dynamic Modules extension.

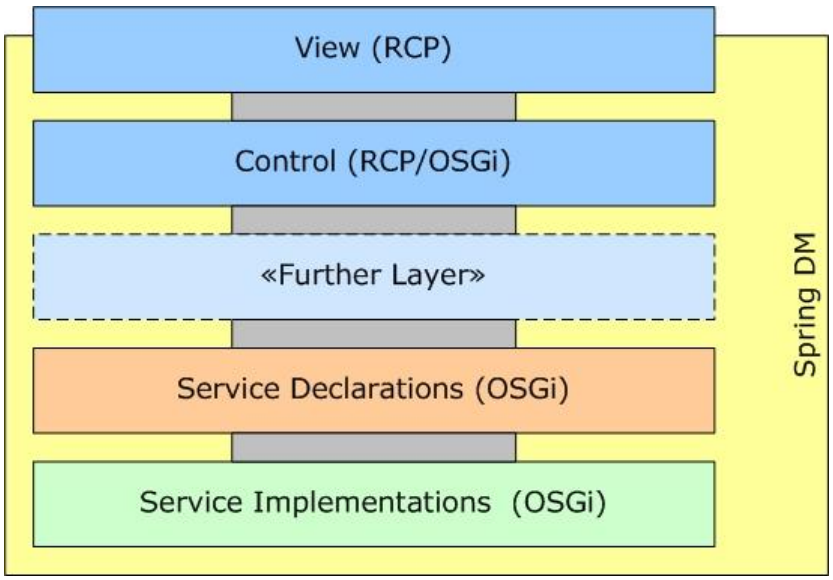

Figure 8: Architecture of an RCP based application with its layers

View

The classes of the view layer implement mainly the user interface with the specific behavior like the event handling and further more. But they do not contain any business logic like the computation of values or the change of objects. User interactions resulting in such a way are forwarded at least to the control layer.

Control

They implement business logic and behavior which is user interface specific. More common business logic is located in the services managed by the OSGi environment.

The classes are in the most cases independent from the used user interface technology in the views, which increases the reusability of them. Only in special cases where such a solution is too complicated or not possible, user interface specific code should be implemented.

«Further Layer»

If required additional layers can be added between the control layer and the services. Their functionality and behavior depends on the specific requirements and design decisions.

Service Declaration

The first part of the service layer is the service declaration. It mainly contains interfaces defining the services as well as its parameters and return values. It contains no implementations because they are located in the service implementation layer, the second part of the service layer.

Service Implementation

The classes of this part of the layer implement the specific service behavior defined by the service declaration. The way these classes are implemented, if they are Web Service clients or local services are part of the design decisions.

6.2.1.1 Text Editor and Graphical Editor Synchronization

Overview

A typical example of such an architecture is the synchronization of a central domain model for different application components. The domain model must be managed central with a common interface to get its values and update it. Interested classes must be notified about changes through an appropriate mechanism.

OSGi service

For the management of the central domain model, we decided to use an OSGi service, like shown in **Figure 9**. The reason for this decision is the fact he must provide a common interface by general and is accessible from all over the application. Through the Whiteboard Pattern *[Kriens04]* it is additionally possible to notify interested counterparts.

They must simple implement defined interfaces and register themselves as services in the OSGi registry. For the notification the notifier can get the currently registered services and notify them. The management of the "listeners" is done by the OSGi framework.

A big advantage of this architecture is its openness. We can add new or remove existing components using the domain models easily. The domain model itself and its management component don't take care about.

UI and control layer

The listeners itself are classes in the control layer, which are responsible to update the corresponding UI component, like a text editor or something else. Additional classes in the control layer are also parsers, which converts user interface specific data in common ones and vice versa to allow a seamless integration.

Generally we tried to hold in these classes the coupling to RCP as low as possible to allow an easy exchange of the UI layer without changes in the controllers.

Schematical class diagram

The following diagram in **Figure 9** shows the architecture of such a management and synchronization of a domain model schematically.

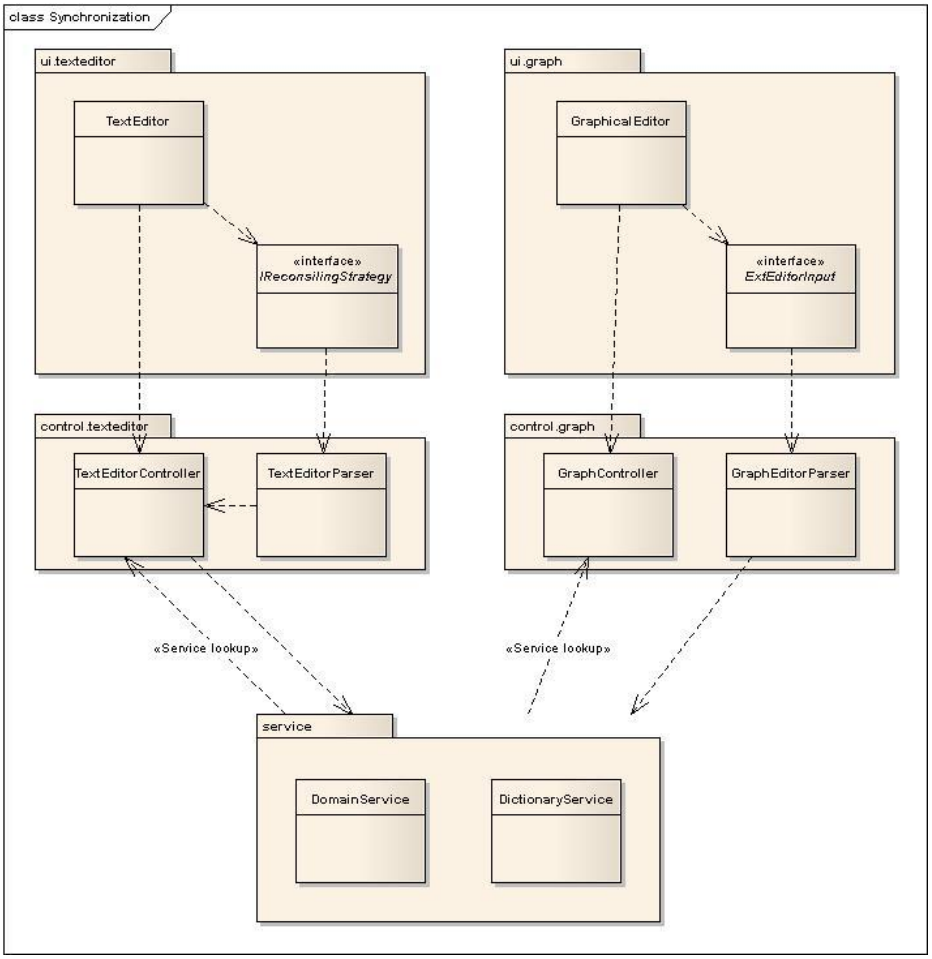

Figure 9: Schematical class diagram for the management of a central domain model

6.2.2 General Topics

Overview

This chapter contains the description of the solution for general architectural topics. These include the naming conventions for bundles and packages.

6.2.2.1 Bundle Naming

Overview

The naming of the different bundles which contains all or specific parts of layer is an important issue. A clear naming is required to distinguish between them.

Solution

In our tutorial applications, we decided to name our bundles in the domain name style like the following example shows. The name consists of different parts describing the organization which wrote the bundle, the application name and the name of the component out of the whole application.

```
org.herasaf.tutorial.calculator.ui
```

The reasons for this decision were on the one hand a clear naming of the bundle to distinguish them and on the other hand to define and show the root package in an easy manner. Therefore it brings mostly a comfortable management and use of the bundles.

On the other side it would have no serious disadvantages if we chose another naming pattern. Only the name of the root package would not be visible from the bundle name, but that's only a discomfort.

6.2.2.2 Package Naming

Overview

The definition of the package names in the different bundles is also an important issue. A clear naming pattern avoids problems in later steps and helps finding the classes in the different bundles in an easy way.

Solution

Based on our solution for the bundle naming, we decided to choose the bundle name also for the name of the root package. It helps on the one hand to locate the classes and packages in an easy way and on the other hand to make problems like split packages between different bundles impossible.

6.3 UI Layer

Overview

In this chapter the solution of all user interface related issues are described. This covers beside the concrete solutions also in particular short analysis and comparisons of alternative approaches and technologies.

6.3.1 Extension Points

Overview

Eclipse provides the Extension Point concept, which enables a modular implementation of an application and the extension through additional bundles, which can be used on demand. In this chapter we describe the use of the extension point concept in our solution.

Intention

In some cases the use of inheritance, aggregation and/or composition is not sufficient in order to create modular applications which can be extended with additional components avoiding the need to change the kernel.

The intention of the extension point concept is to solve the issues with the coupling and the dependencies to enable such a modular design. By following the Hollywood principle "We call you, don't call us" the direction of the utilization is inversed. By using this principle, the core of the application can provide extension points which can be used by other bundles to extend the application without the need to change the behavior in the core implementation.

6.3.1.1 Our solution

Approach

In our solution we use an "empty" RCP application as the core which then is

extended by using the provided extension points.

Extension Points

The core RCP application already includes a set of functionality and provides extension points for its extension. In our solution we made comprehensive use of these extension points for different aspects.

For the definition of views and perspectives we used the extension points `org.eclipse.ui.perspectives`, `org.eclipse.ui.perspectiveExtensions`, `org.eclipse.ui.views`.

The menu- and toolbar actions are defined through the use of the extension points `org.eclipse.ui.menus`, `org.eclipse.ui.commands` and `org.eclipse.ui.handlers`.

The extension points `org.eclipse.ui.editors` and `org.eclipse.ui.editors.documentProviders` are used to define and add editors to the application.

We also used other extension points in our solution. A more detailed description of these can be found in the appropriate chapters in this documentation.

6.3.2 RCP Application

Overview

Developing an RCP application will sooner or later lead to the discussion of how to execute the application. Basically two possible decisions for this discussion exist, depending on specific project and product requirements.

In a first solution, the application will be integrated in Eclipse as a further plug-in, which can be installed if the user needs it. In this case no special configurations and implementations are used and this chapter can be skipped.

The second solution is the execution as an independent application outside of Eclipse. This case requires beside an application generation also some additional configurations and implementations described in detail in this chapter. The generated application consists of only the used components and parts of the Eclipse Platform.

6.3.2.1 Structure

Main plug-in

Like mentioned in the introduction the execution of the application outside of Eclipse requires some additional configurations and implementations. The biggest one is the implementation of an additional plug-in which acts as the main plug-in.

Like the main method in a simple Java application, the main plug-in are the first user written code sequences which will be executed in your RCP application. Therefore it is responsible for the creation of a main window, the initialization of the menu and toolbars and the start of a default perspective.

The implementation of such a main plug-in or the definition of an existing one as the main plug-in is the only way Eclipse RCP allows the definition of independent applications.

Although it is possible to add these classes and configurations in an existing RCP plug-in, but we decided to create a new one to keep the design of the ex-

isting plug-ins clean and focused.

Product definition

Normally it is also the plug-in which contains the product configuration defining general look and feels, dependencies and further more for the whole application.

6.3.2.2 Classes

Overview

Beside the creation of a new plug-in also the implementation of additional classes in it are required as well as its configuration through extension points.

Schematic class diagram

The diagram in **Figure 10** shows the class structure of the additional required classes. A description of them is provided below the diagram.

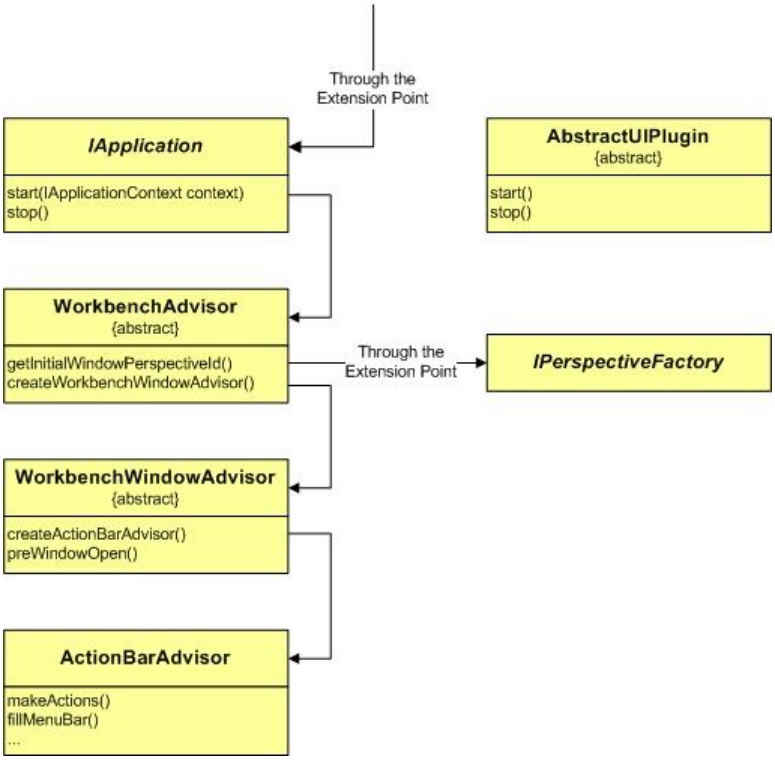

Figure 10: Class structure of a RCP application

Activator

An independent RCP application will start like other plug-ins too an activator extending the `AbstractUIPlugin` class if one exists. It is an optional singleton class which can be used to store and provide global information for the plug-in.

Application

The real entry point to the application defines the extension point `org.eclipse.core.runtime.applications` which registers an application class implementing the interface `IApplication`. This class is responsible for starting and quitting the application.

Alternatively the application class can also be instantiated in the `AbstractUIPlugin`, but this is not a recommended solution. Additionally the direct in-

stantiation increases the dependencies between the two classes unnecessarily compared with the use of the extension point. That's why we implemented our applications with the extension point configuration rather than the direct instantiation.

Listing 7 shows an example of such an application class with the `start` and `stop` methods.

```
public class Application implements IApplication {

  public Object start(IApplicationContext context)
      throws Exception {
    // Create SWT Display
    Display display = PlatformUI.createDisplay();
    try {
      // Create the workbench advisor and run the Eclipse
      // workbench
      int returnCode = PlatformUI.createAndRunWorkbench(display,
          new ApplicationWorkbenchAdvisor());

      if (returnCode == PlatformUI.RETURN_RESTART)
        return IApplication.EXIT_RESTART;
      else
        return IApplication.EXIT_OK;
    } finally {
      display.dispose();
    }
  }

  public void stop() {
    final IWorkbench workbench = PlatformUI.getWorkbench();
    if (workbench == null)
      return;

    final Display display = workbench.getDisplay();
    display.syncExec(new Runnable() {
      public void run() {
        if (!display.isDisposed())
          workbench.close();
        }
      });
  }
}
```

Listing 7: Example of an application class implementing the `IApplication` interface

In the `start` method, the application class starts the *Workbench* component using the `PlatformUI` class. In the `stop` method the workbench must be closed correctly, otherwise the save of the workbench settings and other clean-up work won't be executed.

Workbench

The workbench provides the graphic application framework for the whole application. The start of the workbench will be accompanied by an advisor class, which configures the workbench according the configuration it contains.

This class extends the abstract class `WorkbenchAdvisor` which allows the configuration of the workbench. By overriding the following methods also status-dependent behavior can be executed.

- postShutdown()

- postStartup()
- preShutdown()
- perStartup()

Several further post- and pre-methods exist, but they are deprecated and we didn't use them. So we refrain from describing them here. *[Burnette06]* provides a detailed list of all provided hook methods.

But the main methods are `getInitialWindowPerspectiveId` and `createWorkbenchWindowAdvisor`. The first one, which must be implemented, returns the id of the default perspective which should be shown when starting the application.

The second method, which is optional, creates and returns a workbench window advisor to accompany the creation of the workbench window, the main window of the application. If the method will not be implemented, RCP uses a default advisor. It provides the required functionality like an own one but can be configured only through extension points.

An example implementation of this method is shown in **Listing 8**.

```
@Override
public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
    IWorkbenchWindowConfigurer configurer) {
  return new ApplicationWorkbenchWindowAdvisor(configurer);
}
```

Listing 8: Example of the `createWorkbenchWindowAdvisor` method out of the workbench advisor class

In our applications we implemented the `createWorkbenchWindowAdvisor` method with the corresponding advisor class to configure the workbench window individually. This allows additionally opening editor and other views when starting the application for development and test purposes.

Workbench-Window

The main window in an Eclipse application, also called workbench window, provides the basic buildup given by Eclipse RCP. It can be customized and filled through the application plug-ins implemented by the developer.

Its creation will also be accompanied through a class extending the `WorkbenchWindowAdvisor` class. It is used to configure the status line, menu- and toolbars as well as the title and the size of the window. The **Listing 9** shows an example implementation of such an advisor class.

```
public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {

  public ApplicationWorkbenchWindowAdvisor(
      IWorkbenchWindowConfigurer configurer) {
    super(configurer);
  }

  @Override
  public ActionBarAdvisor createActionBarAdvisor(
      IActionBarConfigurer configurer) {
   return new ApplicationActionBarAdvisor(configurer);
```

```
  }

  @Override
  public void preWindowOpen() {
    IWorkbenchWindowConfigurer configurer =
                                        getWindowConfigurer();
    configurer.setInitialSize(new Point(800, 600));
    configurer.setShowCoolBar(true);
    configurer.setShowFastViewBars(false);
    configurer.setShowPerspectiveBar(true);
    configurer.setShowProgressIndicator(false);
    configurer.setShowMenuBar(true);
    configurer.setShowStatusLine(true);
    configurer.setTitle("<<Application Name>>");
  }
}
```

Listing 9: Example of an advisor class extending the `WorkbenchWindowAdvisor` class

The class provides like the `WorkbenchAdvisor` hook methods to implement custom behavior. The following list shows the most important ones. *[Burnette06]* provides also for this advisor a detailed list of all provided hook methods.

- createActionBarAdvisor()
- postWindowClose()
- postWindowCreate()
- postWindowOpen()
- postWindowRestore()
- preWindowOpen()
- preWindowShellClose()

We mainly used the method `preWindowOpen` to configure the different parts of the application using the `IWorkbenchWindowConfigurer`, like the example in **Listing 9** shows. We used this method because these configurations must be done before the window opens.

Another important method is `createActionBarAdvisor`. It is optional and allows returning an advisor class configuring the action bars programmatically. Alternatively they can also be configured through extension points, whereby this method doesn't have to be overwritten. This was our preferred way, because the configuration through extension points is more flexible and needs less source code.

Action Bar

The advisor class extending `ActionBarAdvisor` allows the registration of actions to add them to the menu-, toolbar and status line. This is the programmatically solution we tried to avoid as much as possible and to do this configurations through the extension point `org.eclipse.ui.menus`. In this case this class is not required and can be left away. This has the advantage of less code to manage and maintain as well as a more flexible configuration of the action bars. Additionally necessary classes to provide extended action bar items can be hooked in easily without adding them manually to them.

A detailed description of the action bar configuration and implementation either in a programmatically manner or with extension points can be found in chapter 6.3.5.

Perspective

The last component of an independent RCP application is the default perspective. In the class extending `ApplicationWorkbenchAdvisor` the method `getInitialWindowPerspectiveId` must return the id of the default perspective. It defines the start screen shown after application finished the startup.

To achieve this, the perspective class must be registered through the extension point `org.eclipse.ui.perspectives` with the used id. Another solution was not possible.

For more detailed descriptions and explanations of the perspective see chapter 6.3.4.

6.3.2.3 Problems

Action Bar

During the development of independent RCP applications we had problem with the action bar advisor class respectively with the action bars. We tried to combine both the programmatically configuration and the one with extension points. The result of these tries were duplicate menu entries and menus one containing the programmatically configured menus, the other the ones configured through the extension points.

We could solve this problem through the assignment of the same ids to the programmatically created menus and menu entries like for those configured with the extension points. This allowed us to use both ways to define action bars and its entries.

6.3.3 RCP Plug-in

Overview

During the development of RCP applications the different components are normally bundled in several plug-ins. RCP makes almost no prescriptions on the structure or the classes and interfaces to provide.

6.3.3.1 Structure

Structure

The RCP plug-ins have a very simple structure, because RCP requires only the existence of the *plugin.xml* and the *Manifest* files. Further prescriptions especially for implementing or extending class or interfaces are not made by RCP.

6.3.3.2 Classes

Activator

An optional component of the RCP plug-in is the activator. It can be used as a singleton like class which provides global information for the plug-in or other static utilities. RCP provides for such an implementation the class `AbstractUIPlugin`, which is an abstract implementation of the `BundleActivator` interfaces.

We used mainly the abstract class for the implementation of own activators, because it provides an extensive set of basic functions. Alternatively the activator can also be implemented only with the `BundleActivator` interface if the

additionally provided functions are not needed.

Plug-in specific classes

The remaining classes and interfaces are totally independent from the plug-in classes. RCP makes no prescriptions for them.

6.3.4 Perspectives & Views

Overview

Generally the user interface of an Eclipse RCP application consists of views. Together with the UI elements, like buttons, fields and furthermore, they contain, they build the smallest unit of a user interface.

Eclipse RCP additionally allows composing whole application sights based on these views, to allow the definition of different user interfaces for different purposes. The advantage of such an aggregation for the user is to switch very fast between these user interface definitions. Additionally the views as the smallest unit can be used in several perspectives.

6.3.4.1 Our Solution

Perspective definition

For the definition of perspectives in our applications, we decided to use the extension point `org.eclipse.ui.perspectives`. It allows a flexible configuration and definition of perspectives. A programmatic definition was in our opinion too rigid. Additionally, we found no descriptions how to define a perspective programmatically.

Beside an id and a name of the perspective, we also had to create a new perspective class inheriting from `IPerspectiveFactory`. In this class we had to implement the method `createInitialLayout` where we could compose the layout of our perspective programmatically.

```
<extension point="org.eclipse.ui.perspectives">
  <perspective class="org.herasaf.tutorial.ui.Perspective"
    id="org.herasaf.tutorial.ui.perspective"
    name="Default">
  </perspective>
</extension>
```

Listing 10: Use of the extension point `org.eclipse.ui.perspectives` for the perspective definition

Perspective composition

We decided not to do a programmatic composition, because it is not flexible enough for further extensions and changes. Additionally the ids of the views are required, with the result that they must be defined also with extension points. So we had to use the extension points anyway. Therefore we decided to use the extension points not only for the definition of the views, but also for composing the perspective.

Eclipse RCP provides for this purpose the extension point `org.eclipse.ui.perspectiveExtensions`. In a first step we had to define which perspective we want to extend and afterwards which views should be part of it. Additionally it's also possible to define perspective and view shortcuts for the corresponding

menu. The following code snippet in **Listing 11** below shows such a declarative composition.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.ui.perspective">
    <view closeable="true"
      id="org.herasaf.tutorial.ui.Navigator"
      minimized="false"
      moveable="true"
      ratio="0.25"
      relationship="left"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      standalone="true"
      visible="true">
    </view>
    <perspectiveShortcut
      id="org.herasaf.tutorial.ui.perspective">
    </perspectiveShortcut>
    <viewShortcut
      id="org.herasaf.tutorial.ui.Navigator">
    </viewShortcut>
  </perspectiveExtension>
</extension>
```

Listing 11: Use of the extension point `org.eclipse.ui.perspectiveExtensions` for the composition of a perspective

View definition

But to use the views for the perspective composition, we had to define and implement them before. For this purpose we used the extension point `org.eclipse.ui.views`. Also here we had to choose either an existing view class or to create a new one, beside the definition of its id and name, like shown in the code snippet of **Listing 12** below.

```
<extension point="org.eclipse.ui.views">
  <view
    category="org.herasaf.tutorial.ui.category"
    class="org.eclipse.ui.navigator.CommonNavigator"
    id="org.herasaf.tutorial.ui.Navigator"
    name="Navigator"
    restorable="true">
  </view>
  <category
    id="org.herasaf.tutorial.ui.category"
    name="Person Locator">
  </category>
</extension>
```

Listing 12: Use of the extension point `org.eclipse.ui.views` for the definition of a view

View implementation

To implement an own view, RCP provides the abstract class `ViewPart`, which should be used as super class. In the method `createPartControl`, which must be implemented, we could add the UI elements to the view and organize them using the passed `Composite` object. Our further implementations were specific to the purpose of the view, why we don't explain them here.

6.3.4.2 Problems

Getting the shell

During the implementation of own views we had the problem that we needed the corresponding `Shell` instance. It is required for example to show dialogs. We tried to get it through the `getAdapter` method but it didn't work fine. After some research and trials we found out we can get it easily through the invocation of `getSite().getShell()`.

6.3.5 Menu- & Toolbar

Overview

The menu and the toolbars are a very important part in order to make the application more useful and practical. They are connected with the different views and parts in the application and provide global actions and commands. In this chapter, we are going to handle the more advanced concepts and show you how we used them in our solution.

6.3.5.1 Global Action Bars

Description

The global action bars and their elements are all the time available during the lifetime of the application. This means, that the global action bar entries are independent from specific views or other parts, which are currently open or active. The global actions, though, can affect views and other parts significantly, like e.g. enabling and activating through an appropriate editor when a file is opened.

6.3.5.1.1 Implementation style

Overview

There exist two ways to fill in the global action bars with elements. The first approach is to implement the menus and toolbar entries programmatically in the program code. The other way is to declare the whole buildup of the menus and toolbars in the *plugin.xml* plugin descriptor.

Alternatives

Alternative 1: Programmatic approach

The definition of the menu and toolbar items can be implemented programmatically in the `ActionBarAdvisor` implementation of the application. In the method `makeActions` the actions are registered, whereas in the `fillXXX` method the items are added to the corresponding action bar.

Advantage	Disadvantage
• All the functionality provided by the concrete implementations is usable. • Complex behavior, like e.g. dynamic creation of items based on domain data, can be implemented.	• Less flexibility through programmatic composition. • Lower reusability of the particular modules.

Alternative 2: Declarative approach

The declarative definition of the menu and toolbar items is done in the *plugin.xml* plugin descriptor. This is done through the use of the extension point `org.eclipse.ui.menus`. With the `menuContribution` clause and the addition of menus (`menu`) and commands (`command`) the items and their order can be specified.

Advantage	Disadvantage
• Makes the application more flexible (e.g. add views to a perspective you didn't create). • Higher reusability (reusable modules) • Matches with the intention and modular nature of the framework.	• Extension points have to be available and provide the required functionality. • It is possible that extension points do not cover all the functionality provided by concrete implementations.

Decision

The question, whether to use a declarative or programmatic approach, also appears in the general UI development in the Eclipse framework. The flexibility and reusability are both very important factors in an evolving application.

According to the advantages of the declarative implementation style, we chose **Alternative 2** for the specification of items to be used in the global action bars.

6.3.5.2 Existing Commands

Description

The framework provides different commands which are already implemented and can be used based on the requirements of your application. In this chapter we describe how we used the existing commands in the framework for our solution.

6.3.5.2.1 Our Solution

Used commands

In our solution we use the following predefined commands in the Eclipse framework:

- Save (`org.eclipse.ui.file.save`)
- Quit (`org.eclipse.ui.file.exit`)
- Undo (`org.eclipse.ui.edit.undo`)
- Redo (`org.eclipse.ui.edit.redo`)
- Cut (`org.eclipse.ui.edit.cut`)
- Copy (`org.eclipse.ui.edit.copy`)
- Paste (`org.eclipse.ui.edit.paste`)
- Reset Perspective (`org.eclipse.ui.window.resetPerspective`)

The framework provides a lot more predefined commands, which can be made visible by pressing the *"Browse…"* button in the plugin development environment in Eclipse.

6.3.5.3 Custom Commands

Description

In many cases, where a custom behavior of a command is required and when the framework does not provide a default implementation, a custom command handling has to be implemented. In this chapter we describe how we developed custom commands in the framework.

6.3.5.3.1 Implementation style

Overview

The handling of the commands can be realized in two ways. The first approach is to create actions and link them to commands. The other way is to create a handler for the command.

Alternatives

Alternative 1: Use actions for the command handling

On the extension point `org.eclipse.ui.editorActions` an `editorContribution` can be defined, which contains the concrete actions. An action has to implement the interface `IEditorActionDelegate` in order to be recognized as an action in the framework. The link to a command, which makes an action act as a handler for the particular command, is established through the attribute `definitionId` in the action clause (`action`). The concrete implementation of the action is realized in the `IEditorActionDelegate` implementation class.

Advantage	Disadvantage
• It is possible to react on selection changes in the editor (`selectionChanged` method). • It is possible to react on editor activation changes (`setActiveEditor` method). • Comfortable use and lower implementation effort.	• Different extension points exist for the contribution to different workbench parts (popup menu, editor, toolbar, etc.). This leads to duplicate XML code in the *plugin.xml* descriptor. • Duplicating the actions leads to multiple instances of the same action in the memory. • The action is tied in with a specific editor through the `editorContribution`. This increases coupling and reduces the reusability. • The flexibility is affected through the specification of actions in multiple extension points. • Depending on the implementation, dependencies to the editor can occur in the action.

Alternative 2: Use handlers for the command handling

Through the use of the extension point `org.eclipse.ui.handlers` handlers (`handler`) can be specified to handle a particular command. A handler is specified to implement the interface `IHandler`. The clauses `activeWhen` and

`enabledWhen` allow configuring the visibility and activation of the handler.

Advantage	Disadvantage
• A command is the abstract idea of some code being executed. • The declaration of the command and the handling are independent from each other. Thus, a command can have multiple handlers. • Avoids duplicating the XML code and increases flexibility. • The coupling is held low and the reusability high.	• More implementation effort. • Reaction on editor changes and selection changes is more complicated to realize.

Decision

Based on the overbalance of advantages we decided to use handlers for the command handling (**Alternative 2**). Additionally the handlers are also the more modern concept, why it is recommended to use it.

6.3.5.3.2 Our Solution

Overview

Based on the decisions above, we use the declarative approach for the implementation of the menu and toolbar items and handlers for the handling of the commands. In this chapter we describe our solution concerning the menu and toolbar.

Custom commands

For the definition of custom commands we use the extension point `org.eclipse.ui.commands` in our solution. There are already existing commands provided by the framework, which can be reused in many cases. However, creating custom commands for specific needs requires the declaration in the mentioned extension point and the implementation of an appropriate command handler. Command handlers are specified using the extension point `org.eclipse.ui.handlers`. A handler (`handler`) assigns a class implementing the interface `IHandler` to a command, which is invoked every time the command is executed.

The following code snippet in **Listing 13** shows the specification of custom commands in an example application.

```
<extension
  point="org.eclipse.ui.commands">

  <command
    id="org.herasaf.tutorial.personlocator.ui.dictionaryCommand"
    name="Dictionary">
  </command>
</extension>

<extension
  point="org.eclipse.ui.handlers">
```

```
  <handler
    class="org.eclipse.springframework.util
       .SpringExtensionFactory:dictionaryHandler"
    commandId="org.herasaf.tutorial.personlocator
       .ui.dictionaryCommand">
  </handler>
</extension>
```

Listing 13: Snippet of the specification of custom commands in `plugin.xml`

Action bar items

For the creation of the main menu in the application we used the extension point `org.eclipse.ui.menus`. By defining a `menuContributon` with the `locationURI` *"menu:org.eclipse.ui.main.menu"*, contributions to the main menu are realized. As for the commands (`command`) in the `menuContribution`, we use existing commands provided by the framework for actions like save, cut, copy, paste, undo, redo and exit.

```
<extension
  id="fileEnd"
  point="org.eclipse.ui.menus">
  <menuContribution
    locationURI="menu:org.eclipse.ui.main.menu">
    <menu
      id="file"
      label="&amp;File">
      <separator
        name="fileStart"
        visible="false">
      </separator>
      <command
        commandId="org.eclipse.ui.file.save"
        id="save"
        mnemonic="S"
        mode="FORCE_TEXT"
        style="push">
      </command>
      ...
  </menuContribution>
</extension>
```

Listing 14: Snippet of the specification of menu bar items in `plugin.xml`

For custom items we used the extension point `org.eclipse.ui.commands`, allowing the definition of commands, and `org.eclipse.ui.handlers` for the implementation of command handlers. The behavior of the command is implemented in the `execute` method of the command handler (`IHandler`).

For the contribution of items to the toolbar we use a `menuContribution` with the `locationURI` *"toolbar:org.eclipse.ui.main.toolbar"*. The addition of menus (`menu`), separators (`separator`) and commands (`command`) is done the same way as for the main menu.

The menu items for the selection of the perspective and views are implemented programmatically in the `ActionBarAdvisor` of the application. In the method `makeActions` a perspective short list is gathered through the use of the `ContributionItemFactory` and the `IWorkbenchWindow`. This list is then assigned to a `MenuManager` item and added to the menu bar in the method `fillMenuBar`. A declarative solution could not be acquired in the scope of this thesis.

The code snippet in **Listing 15** below shows our solution in an example application.

```
protected void makeActions(IWorkbenchWindow window) {
  super.makeActions(window);

  ...
  perspectiveShortList = ContributionItemFactory
                         .PERSPECTIVES_SHORTLIST.create(window);
   viewShortList = ContributionItemFactory
                   .VIEWS_SHORTLIST.create(window);
}

protected void fillMenuBar(IMenuManager menuBar) {
  super.fillMenuBar(menuBar);

  ...

  MenuManager windowMenu =
    new MenuManager("&Window",
                    IWorkbenchActionConstants.M_WINDOW);
  menuBar.add(windowMenu);

  MenuManager perspectiveMenu =
    new MenuManager("&Open Perspective");

  perspectiveMenu.add(perspectiveShortList);
  windowMenu.add(perspectiveMenu);

  MenuManager viewMenu = new MenuManager("Show &View");
  viewMenu.add(viewShortList);
  windowMenu.add(viewMenu);

  ...
}
```

Listing 15: Snippet of the action bar implementation in the `ActionBarAdvisor` class

Shortcuts

To make the use and the selection of views and perspectives in the application more practical, shortcuts can be defined. A shortcut is shown in the perspectives or the views menu to simplify their access.

Through the use of the extension point `org.eclipse.ui.perspectiveExtensions` a `perspectiveExtension` is created, which contains the `perspectiveShortcuts` and `viewShortcuts` to the appropriate items.

The following code snippet in **Listing 16** shows the definition of shortcuts in the person locator example application.

```
<extension
  point="org.eclipse.ui.perspectiveExtensions">

  <perspectiveExtension
    targetID=
      "org.herasaf.tutorial.personlocator.ui.perspective">
    ...

    <perspectiveShortcut
      id=
        "org.herasaf.tutorial.personlocator.ui.perspective">
```

```
    </perspectiveShortcut>

    <viewShortcut
      id=
        "org.herasaf.tutorial.personlocator.ui.Navigator">
    </viewShortcut>
  </perspectiveExtension>
</extension>
```

Listing 16: Snippet of the configuration of shortcuts for views and perspectives

6.3.5.4 Problems

Incorrect order of the menus

By defining the menu items using the declarative approach in combination with the programmatic one, we had the problem that the order of the items was not as specified in the plugin descriptor. The cause of this problem was that the programmatic composition was evaluated before the declarative configuration.

We solved this problem by adding all the menus in the `ActionBarAdvisor` implementation class programmatically, like shown below.

```
protected void fillMenuBar(IMenuManager menuBar) {
  super.fillMenuBar(menuBar);

  menuBar.add(new MenuManager("&File",
                              IWorkbenchActionConstants.M_FILE));
  menuBar.add(new MenuManager("&Edit",
                              IWorkbenchActionConstants.M_EDIT));
  ...
}
```

Listing 17: Snippet showing the solution of the menu item order problem

The problem with this solution is that the definition of menu items now exists in the *plugin.xml* descriptor as well as in the `ActionBarAdvisor` implementation.

Global action bar entries not active

Another problem we had during the implementation of the menu and toolbars was that after the declarative definition of the workbench actions like cut, copy, paste, etc., the entries were not active. A detailed description of this problem and the solution we chose can be found in chapter 0.

6.3.6 UI elements

Overview

Important parts of the UI layer are the UI elements. They build the visible part of an application and bring therefore own challenges and problems. In this chapter, we are going to handle the more advanced concepts and show you how we used them.

6.3.6.1 UI element implementation style

Intention

The problem to reduce the amount of duplicate code is, especially in the UI

programming, a huge challenge. Nothing else uses as much inner classes, temporary variable and duplicated code as UI classes. To allow an easy maintenance and extension at any time, the reduction of duplicate code is a main issue.

To allow a fast and clean implementation of the UI classes each situation needs its own solution.

6.3.6.1.1 Implementation styles

Overview

To reach a reduction of duplicate UI code and an increased maintainability several solutions exist, which we compare in the comparison below. Which solution the best is depends largely on the concrete situation.

Alternatives

Alternative 1: Use the raw UI classes

The most obvious solution is to use the provided UI classes in its raw form. Additional functionality like the registration of listeners or the set of options will be made outside the UI class itself if required.

Advantage	Disadvantage
• The existing inheritance hierarchy will not be increased through the derivation of own UI elements. • No additional classes are used for the implementation of the UI. • If the UI element needs no or only a few configurations, it's an appropriate and simple solution.	• Recurring configurations and options have to be made many times with duplicated or very complicate program code. • The developer has to consider not forgetting a configuration, e.g. to register a listener or other options, when he adds additional UI elements or change the existing ones.

Alternative 2: Create own UI classes through derivation

Another very popular solution is the extension of the existing UI classes through derivation. Recurring and common code sequences can be moved into this new class and has to be implemented only once. Only the individual functionality like register listeners or the set of options will be made outset the UI class.

Advantage	Disadvantage
• Common code sequences are implemented only once. This increases the maintainability of the application. • Through the derivation the configuration of the UI class can be hidden from the user	• The implementation of additional derived UI classes can bloat the design. • The functionality of the general UI classes might be limited. But restricting the super class conflicts with the Liskov substitution prin-

of the class. He doesn't have to consider them. • The forum members *[Eclipse-ZoneSubclassing]* suggest this solution only under the restriction to use only the public accessing methods.	ciple[5]. • It is not worth to create a derived class for each usage, if there aren't common parts which can be moved to them. This would bloat the design disproportionately. • This solution is not suggested by the Eclipse SWT newsgroup *[SWTSubclass]*, because it may lead to dependencies to the platform specific implementation of the UI classes. • The direct subclassing can lead to a limited portability between the platforms (Windows, Linux, etc.). • Additionally the `checkSubclass` method must be overwritten to prevent the subclass check.

Alternative 3: Create a wrapper for UI classes

A third alternative consists of a wrapper which encapsulates the access to the concrete UI class. Therefore it is a matter of the *Decorator Pattern* described by the Gang of Four *[GoF1995]*.

The wrapper contains an instance of the concrete UI class and provides methods for its access. The most of them pass the calls to methods of the UI element, only the ones which extend the functionality implement further statements.

Additionally the wrapper must derive either from the *Canvas* or *Composite [SWTSubclass]* class to provide a minimal functionality required for UI elements. Otherwise the wrapper must override too many methods. A tutorial how to implement such a wrapper is provided by the Eclipse Corner Article Site *[Northover01]*.

Advantage	Disadvantage
• Common code sequences are implemented only once. This increases the maintainability of the application. • Through the use of a wrapper, the configuration of the UI class can be hidden from the user of the class. He doesn't have to consider them. • This is the propose solution of the Eclipse SWT newsgroup *[SWTSubclass]* for the extension of UI classes, because it prevents dependencies to the	• The implementation of too much wrapper classes can bloat the design. • The use of simple wrappers can also bloat the design, because they forward all or the most method calls to the methods of the wrapped UI element. • It adds an additional indirection. This may lead to higher space consumption and to a slower execution.

[5] Liskov substitution principle: http://en.wikipedia.org/wiki/Liskov_substitution_principle

platform specific implementation of the UI classes.

- The wrapper allows the unlimited portability between the different platforms (Windows, Linux, etc.).

Decision

The comparison of the alternatives above shows the dilemma of this issue, because a correct solution does not exist. It depends on the concrete situation which solution is the most appropriate one.

In our solution, we decided to use on the one hand the raw UI classes and on the other hand the wrappers for their extension.
The raw UI classes are for us the most appropriate choice in conventional user interfaces. In these, the different types of the UI classes vary greatly. For this reason the implementation of own subclasses or wrappers would not bring any advantage.

Different the wrappers, they are for us the most appropriate solution in cases where the concrete user interface consists of a lot of very similar UI elements. In this case, it can lead to a simplification of the development as well as the maintenance of the implemented user interface. To mention an example, imagine a calculator application with a numpad. The number buttons vary only in the number they represent. The other configurations and options are the same, so we can easily centralize them in a wrapper.

The implementation style using the subclassing of existing UI classes is for us not an appropriate solution. We tried to avoid this implementation style whenever possible, because the risk of using non-public attributes and methods is too high. This is required to develop a subclass of an UI class, which is portable without any problems.

6.3.6.1.2 Used implementation style

Raw UI elements

In out implementations we mainly used the raw UI elements for the implementation of our graphical user interfaces, because they are used in different ways with own configurations and options.

How to implement a user interface with the raw UI elements is very simple. You can find a lot of tutorials and descriptions in books and websites. That's why we don't explain it here.

Wrappers

In special user interfaces, where we used a lot very similar UI elements, we decided to use wrapped elements. They helped us to centralize the common parts of the configuration so we had to implement it only once. Additionally this allowed also the extension of the UI element with specific behavior.

For its implementation we inherited from `Canvas` and overrode the necessary methods. Important was in this case the constructor where we added the contained elements using the passed `Composite`.

Its usage in the user interface was afterwards the same like using a conventional UI element. Only the name of it was different.

6.3.6.1.3 Problems

SWTException

During the study and trial of the different implementation styles for UI elements, exceptions occurred while implementing subclasses of existing SWT elements. The reason for this error was the method `checkSubclass`, which contains a check if the current class is a subclass. If the current class is a subclass, the method throws an exception, because SWT don't want the developers to implement subclasses of the elements. For more details see Eclipse SWT newsgroup FAQ *[SWTSubclass]* about this issue.

We had to override the mentioned method to avoid the execution of the check. But this solution is not recommended.

6.3.7 User Layout

Overview

The Eclipse framework gives a lot of functionality concerning the configuration and the positioning of views and other components on the user interface. It also provides a feature which saves user specific preferences at close and restores them on startup.

6.3.7.1 Our solution

Save and Restore

In order to enable the save and restore feature provided by Eclipse the method `initialize` has to be overridden in the `WorkbenchAdvisor` implementation. The save and restore feature is enabled through the `IWorkbenchConfigurer`.

The following code snippet shows the content of the `initialize` method in an example application.

```
public void initialize(IWorkbenchConfigurer configurer) {
  super.initialize(configurer);
  configurer.setSaveAndRestore(true);
}
```

Listing 18: Activation of the save and restore feature in the `WorkbenchAdvisor`

6.3.8 Graphical Editor

Overview

The use of a graphical editor is an important issue in today's applications for visualization and manipulation purposes. Therefore such an implementation has to satisfy several requirements, like an easy development and extension, a flexible structure and further more.

The *Eclipse Rich Client Platform* provides for such graphical editors different solutions and frameworks, each with its individual advantages and disadvantages.

In this chapter we take a detailed look at the different RCP frameworks and compare them to come to a clear decision which is the appropriate one we want to use.

6.3.8.1 Technologies

Intention

Eclipse provides for graphical visualization and manipulation two frameworks, the *Graphical Editing Framework (GEF)* and the *Graphical Modeling Framework (GMF)*, whereas the second bases upon the first one.

GEF is a framework providing abstract and basic classes for the implementation of own graphical editors whereas GMF provides the developer with different models to generate a graphical editor with them.

Both approaches have individual advantages and disadvantages we are going to compare in the following comparison to decide which technology we want to use. The decision for the wrong one can increases the analysis and implementation effort immensely what we try to prevent.

Alternatives

Alternative 1: Graphical Editing Framework (GEF)

The *Graphical Editing Framework [GEF]* provides, like mentioned above, common implementations and abstract classes to hook-in for the development of own graphical editors. The framework exists since the Eclipse version 2.1 published in March 2003 and has refined since then. It has proven as a useful and stable technology.

Advantage	Disadvantage
• GEF allows implementing individual characteristics of the graphical editor *[Lee03]*, only hard to achieve with GMF. • GEF provides a wide range of different features to develop and extend the graphical editor. *[Daum2008]* • Through the own implementation, the integration in an existing application is easy to achieve. • Also the integration in a build environment like Maven is also easy to achieve, because no projects are generated or overridden. • The use of the *Eclipse Modeling Framework [EMF]* or EMF for short, to generate a diagram model is possible but not required like in GMF. *[Aniszczyk05]* • The GEF classes and interfaces are provided by only two plug-ins. Therefore the use of GEF increases the dependencies marginally.	• Although the implementation model of GEF is simple, it requires a comprehensive knowledge. • The whole implementation has to be made manually, which requires a higher effort compared with the code generation in GMF. • Advanced features like an automatic arrangement of the elements or others are not provided by GEF. They had to be implemented manually. • Depending on the visualization of the domain model, a special diagram model is required which can hold the diagram specific information like connections, positions, sizes and further more. • In some cases the concepts are not generic enough to be used in a wide range of applications. *[Daum2008]*

- In the internet and through books comprehensive tutorials and descriptions are available. *[GEFDevGuide]*
- The implementation model behind GEF is very simple and easy to understand. *[GEF]*

Alternative 2: Graphical Modeling Framework (GMF)

The *Graphical Modeling Framework [GMF]* is not a totally new framework; it is much more a combination of the existing *Eclipse Modeling Framework [EMF]*, or EMF for short, and the *Graphical Editing Framework [GEF]*. It uses both technologies and combines and extends them with further functionalities.

Additionally it provides a new approach for the development of the graphical editors. Instead of writing code, the developer designs models for the diagram model, the visualization and the tooling and generates the diagram code out of them. The **Figure 11** below shows the workflow for diagram editor generation.

In a first step the domain model for the diagram will be designed and generated using the *Eclipse Modeling Framework*. Based on it the model for graphical representation and the tooling will be created.

In further step the three models will be mapped to allow the generation of a Generator Model, whose purpose is to generate the final diagram editor plug-in.

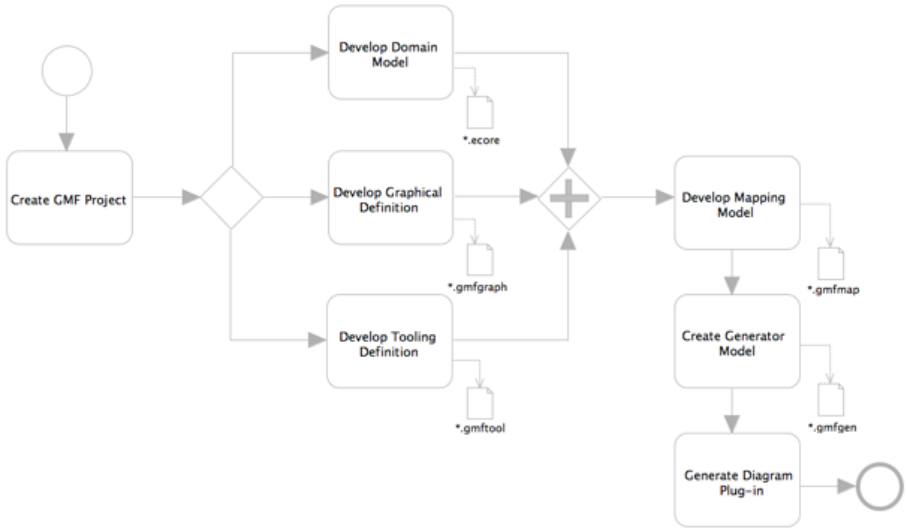

Figure 11: Workflow of GMF for the generation of an diagram editor (Source: *[GMFTutorial]*)

The framework exists since the Eclipse version 3.2 published in June 2006 and has been refined since then. It is a stable technology.

Advantage	Disadvantage
• GMF allows through the derivation of the models among	• Although GMF abstracts the diagram editor development through its

each other and the generation of code based on them, a development of standard diagram editors with less effort.

- The program code is generated, and must not be implemented manually.
- It provides an extensive number of advanced features like an automatic arrangement of the elements and furthers more.
- Changes and customization in the generated code base can be marked with `@generated NOT` in the javadoc to prevent its overwriting. *[GMFGraphHints]*
- The diagram model generated with EMF provides the required fields and functions for notification and synchronization by default.

model concept, the programmers need some knowledge about the structure and behavior of GEF to understand the configurations.

- The diagram editor is not generic, because it is designed and generated for a specific model.
- Changes and customization can be made only at method level. Marks for finer-grained changes cannot be made. *[GMFGraphHints]*
- GMF requires an own diagram model generated by EMF. An own model, like a general domain model cannot be used. Therefore a conversion is mandatory for a synchronization with an own model. *[GMFTutorial]*
- The generated diagram editor uses by default files for saving the diagram data. To change the diagram data source requires immense refactorings and changes.
- GMF generates a large amount of classes (about 10'000 lines of code for a simple editor) and uses also a lot of extension points. To find hook-ins is for a customization is very hard to achieve.
- For GMF only few tutorials and descriptions exist which are up-to-date and helpful.
- GMF generates not only source code, but also whole plug-in projects. An integration of them in build tools like Maven must be made manually.
- An automatic generation of the diagram editor through a build tool is not possible. The reason is on the one hand configurations which must be made manually and on the other hand configurations which must be checked very time to avoid wrong links between the tooling, the visualization and the diagram model.
- Compared with GEF, GMF consists of a large amount of own plug-ins. Additionally it requires an extensive number of further plug-ins either for the generation or for the execution of the diagram editor.

	• To configure the models for the code generation properly, their configuration purposes must be learned, especially their specifics. But such descriptions are rarely. *[GMFGraphHints]* • The installation of the development environment, especially in combination with the Q4E/IAM plug-ins, is cumbersome. • Also the setup of the target platform is very cumbersome, cause of the dependencies to other plug-ins. • For read-only visualizations, GMF is too heavy-weight. • The generated code leads additionally to hundreds of warnings, because the code generator uses raw types instead of generic ones.

Alternative 3: Extended Graphical Editing Framework (GEF++)

A third alternative for the development of a graphical editor the use of an extended GEF, or GEF++ for short. This means primarily to use the *Graphical Editing Framework* as a basis and to extend it either manually or with the use of GMF components with missing functionalities.

Advantage	Disadvantage
• GEF allows implementing individual characteristics of the graphical editor *[Lee03]*, only hard to achieve with GMF. • Through the own implementation, the integration in an existing application is easy to achieve. • Also the integration in a build environment like Maven is also easy to achieve, because no projects are generated or overridden. • The use of the *Eclipse Modeling Framework [EMF]* or EMF for short, to generate a diagram model is possible but not required like in GMF. *[Aniszczyk05]* • In the internet and through books comprehensive tutorials and descriptions for GEF are	• An extension of GEF with a selected number of GMF plug-ins isn't documented until now. No descriptions about such an approach are available. This is a high risk. • Compared with GEF, its implementation requires a more comprehensive knowledge. • The whole implementation has to be made manually, which requires a higher effort compared with the code generation in GMF. • Depending on the visualization of the domain model, a special diagram model is required which can hold the diagram specific information like connections, positions, sizes and further more.

available and can be used for its extension.

- The generated diagram editor from GMF can be used as examples.
- Advanced features may be used directly from the GMF libraries.
- The implementation model behind GEF++ is the same as behind GEF.
- No problems, which may occur through the code generation, exist.
- The setup of the development environment including the target platform is easier compared with GMF.

Decision

Based on the comparison of the different technologies above, we decided to use the *Graphical Editing Framework* for the visualization and manipulation of domain model data.

The reason for this decision is mainly the poor adaptability of GMF and the complex configuration of its models with hundreds of options. Additionally the integration of the diagram editor generation in the whole development process including the automatic build, the development environment and the target platform is cumbersome. All in all GMF provides several simplifications and features, but brings also huge disadvantages which overbalance all the advantages.

Also the GEF++ as an extension of the GEF with selected GMF plug-ins is not a satisfactory solution. The risk of incompatibilities and problems is very high compared with the few additional features and possibilities it has.

Therefore the disadvantage of GEF with the manual implementation and the higher effort to achieve this, is compared with its advantages an acceptable compromise. Additionally it allows a GEF++ like development through a generic implementation of new features in order to provide them as a further plug-in to the existing ones.

6.3.8.2 The Editor

Overview

The editor is the core of the graphical editor component. Without the editor neither the visualization nor a manipulation of the diagram and the model behind it is possible. But the visualization and its manipulation is only a part of its responsibilities. It is also responsible to provide the user with a palette, an outline, key handlers, a zoom manager and further more.

Additionally, it has to support state-dependent behavior like showing the dirty state, providing the save function in such a way or the handling with the command stack.

Schematic class diagram

The class diagram in **Figure 12** shows the dependencies between the different parts of the graphical editor schematically, with the `GraphicalEditor` in its center.

The `GraphicalEditor` is an abstract class providing the base functionality of a graphical editor together with the `GraphicalViewer`. The editor class is mainly responsible for creating the required classes for the used components. Different the viewer class which is responsible for the visualization of the diagram and is fully provided by GEF. For the development of an own graphical editor, `GraphicalEditor` or a derivation of it must be used.

The graphical editor either creates and initializes or uses the parts displayed around it and the graphical viewer, like the `EditDomain` responsible to hold the data and handles the interactions made with the graphical editor. One of the classes it holds is the `PaletteViewer` which is created by the `GraphicalViewer` using the provided `PaletteRoot`.

The `IEditorInput` is an interface describing some kind of input data for the editor. This can be the data saved in a file or managed by a central component. He's responsible to supply the editor with input data even if they are empty or still does not exist.

A further part is the `IContentOutlinePage` as the base interface over the specific implementation. They allow displaying an outline view of the editor data, like known from the Eclipse Java Development Environment.

Different is the `ScalableRootEditPart`, created by the graphical editor but used by the graphical viewer. It is used in combination with the `ZoomManager` to provide zoom functionality for the editor. Otherwise this feature must be implemented on your own.

Similar the `GraphicalViewerKeyHandler` respectively its super class `KeyHandler`, which defines key handlers for alternative interactions with the user, compared with the palette. They allow the creation of elements and their manipulation.

Also an alternative form of interactions provides the `ContextMenuProvider`. It is created by the graphical editor as well and allows the usage of a context menu in the graphical viewer.

The last and most important part of the graphical editor is the `EditPartFactory` and the corresponding `EditPart` classes used by the graphical viewer. The factory is needed to create the edit parts for a given model object. An edit part acts therefore as the mediator between the `IFigure` representing the view it creates and the model. In the *MVC-Pattern [MVC07]* it is also known as controller. They build the core of the graphical editor because they hold the real data, visualize them and control the interactions on them.

Therefore they use the edit policies which handle the interaction requests and create `Command` objects for the action execution. This covers also the interactions through the palette and the key handlers.

The `Command` objects are on the one hand used for the execution of the different manipulations in a similar manner and on the other hand to provide the Undo/Redo functionality.

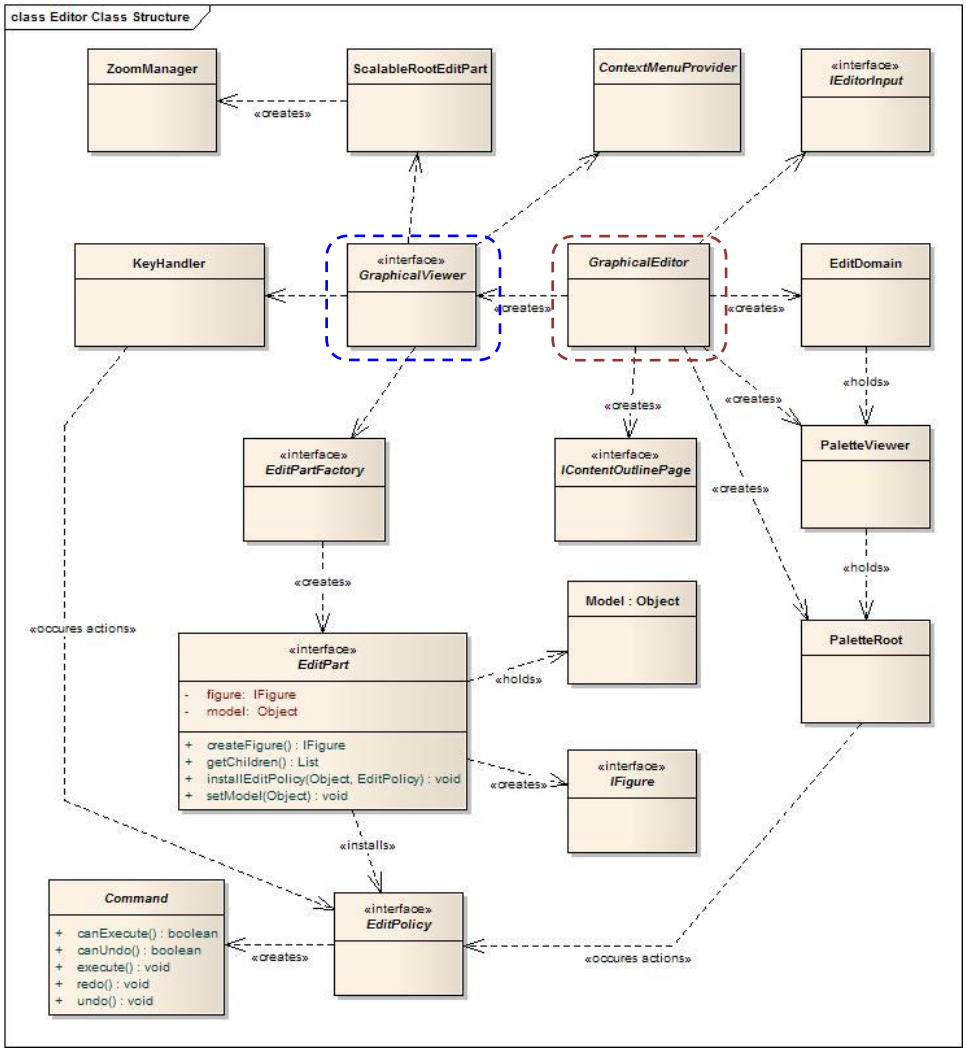

Figure 12: Schematic structure of the classes and interfaces in the graphical editor

In the following chapters we take a detailed look at the different parts of the graphical editor. Therefore we give here no more explanations to them.

6.3.8.2.1 Our Solution

Editor class

In our solution we decided to inherit our graphical editor class from `GraphicalEditorWithFlyoutPalette` instead of the abstract class `GraphicalEditor`, because it provides amongst other things an integrated palette support and is easier to use. The use of the `GraphicalEditor` as the super class would be also possible, but cause implementation overhead for functions already provided by the subclasses.

We additionally decided to implement our graphical editor divided into two classes, where one is an abstract class which contains the general parts of the implementation. This allows us to minimize the implementation effort for the specific editors.

The super class forced us to implement the two methods `doSave` and `getPa-`

letteRoot. In a first step we let them empty and returned null or another default value.

To integrate the editor in the RCP application it was necessary to register our class through the extension point org.eclipse.ui.editors. Otherwise the component who wants to show something in the graphical editor must add it to the RCP application. This causes direct dependencies between the graphical editor plug-in and the invoker, which should be prevented as much as possible. Additionally such an implementation has a bad design.

Configuration & Initialization

For the initialization of the opening graphical editor, we overrode in the init method of our class and invoked the setEditDomain method. We passed a new instance of the DefaultEditDomain, which is responsible to handle all the interactions with the graphical editor in the background and to hold the required data for this purpose. **Listing 19** shows a code snippet with the init method used in a tutorial application.

We also could set the EditDomain in the constructor of the class, which works fine. This was also our first solution, but we had to move the method call later to the init method, because of problems with the Spring wiring (see chapter 6.3.8.2.2).

```
@Override
public void init(IEditorSite site, IEditorInput input)
        throws PartInitException {
  setEditDomain(new DefaultEditDomain(this));
  super.init(size, input);
}
```

Listing 19: `init` method of the graphical editor with the new `DefaultEditDomain`

Further we overrode the method configureGraphicalViewer to extend the configuration before the content of the graphical editor will be set. An important configuration is for example the registration of the EditPartFactory (see chapter 6.3.8.4.1.1). Its configuration will be shown in the following chapters, because they depend on them directly. **Listing 20** shows a code snippet with the overridden method.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  // Further configurations
  ...
}
```

Listing 20: Example of the method `configureGraphicalViewer` in the graphical editor class

After the invocation of the constructor and the configure method, the method initializeGraphicalViewer will be invoked. Following the Javadoc of the method, its main purpose is to set the contents of the GraphicalViewer after it has been created. This was also the only thing we did in this method. **Listing 21** shows a code snippet with this method. The used variable is an object which

was created in a prior step.

```
@Override
protected void initializeGraphicalViewer() {
  GraphicalViewer viewer = getGraphicalViewer();
  viewer.setContents(contents);
}
```

Listing 21: Example of the method `initializeGraphicalViewer` in the graphical editor class

6.3.8.2.2 Problems

Dirty state

A strange problem we had in the graphical editor was the visualization of the dirty state. After the manipulation of the diagram, the state changed but was not shown in the tab, for example with a *.

The cause of this problem was the missing notification of the user interface to show it. Therefore we had to override the method `commandStackChanged` in our graphical editor class and to fire the `PROP_DIRTY` notification, like shown in **Listing 22**.

```
@Override
public void commandStackChanged(EventObject event) {
  super.commandStackChanged(event);

  firePropertyChange(PROP_DIRTY);
}
```

Listing 22: Override of the `commandStackChanged` method to solve the dirty flag problem

EditDomain

During the switchover from the direct instantiation of the used components to the Spring wiring, we had problems with the `EditDomain`. While the `EditDomain` is created and set, the graphical editor invokes the method `getPaletteRoot` to fill in the `PaletteRoot`. But to this time, the `PaletteRoot` was not injected yet. To set the `EditDomain` in the `configureGraphicalViewer` method is on the other hand too late, because the graphical editor needs it before.

So we found the possibility to override the `init` method and to set the `EditDomain` there. This is late enough to allow Spring finishing its injections but also early enough to set it before it is used.

EditDomain creation

Another problem occurred in connection with the `EditDomain` was a `NullPointerException`. Opening the graphical editor a first time worked very well but reopening it in a later moment unexpectedly not.

The cause of this problem was the combination of RCP with Spring. Instead recreating the application context when opening the graphical editor, RCP seems to reuse an already existing instance. In this case the `EditDomain` is already set and don't have to be overridden by a new instance, like we implemented in the `init` method.

To solve this problem, we checked in the method `init` if the `EditDomain` already exists. So we created a new instance only in the case where now one

exists. The code snippet in the **Listing 23** shows the improvement in the `init` method.

```
@Override
public void init(IEditorSite site, IEditorInput input)
        throws PartInitException {
  if (getEditDomain() == null) {
    setEditDomain(new DefaultEditDomain(this));
  }
  super.init(size, input);
}
```

Listing 23: Improved `init` method of the graphical editor with the new `DefaultEditDomain`

Another and more recommended solution was the configuration of the beans in the Spring application context with the scope *prototype*. In this case every time the bean is requested, Spring creates a new instance of it.

6.3.8.3 The Editor Input

Overview

An important part of the graphical editor is the editor input. It must be passed to the graphical editor in a general manner but open enough to provide several implementations.

Eclipse RCP provides for this purpose the `IEditorInput` interface for the development of specific editor inputs. The Javadoc describes the `IEditorInput` with the following statement: *"The `IEditorInput` is a light weight descriptor of editor input, like a file name but more abstract. It is not a model. It is a description of the model source for an `IEditorPart`, an interface implemented by the `GraphicalEditor` class."* The reason for this design lies in the navigation history of the workbench. *"It tends to hold on to editor inputs as a means of reconstructing the editor at a later time. The navigation history can hold on to quite a few inputs (i.e., the default is fifty)."*

6.3.8.3.1 Our Solution

Extended interface

In our solution we also used the interface `IEditorInput`, because it is the only way to pass different inputs to the graphical editor in a RCP-compliant manner. But we extended the interface through an own interface with additional methods to get the input data and to save it. The reason for this decision was to provide an interface instead of a concrete class with methods to get and save the editor data. This allows an easy exchange of the concrete implementation without changes in the graphical editor. Otherwise the graphical editor would have a dependency to a specific `IEditorInput` implementation, which should be prevented in an extendable and flexible application.

Listing 24 shows our extended interface with these two methods.

```
public interface IExtEditorInput extends IEditorInput {

  public Object getEditorInput();
```

```
  public void save(Object editorData);
}
```

Listing 24: `IExtEditorInput`, an extension of the `IEditorInput` with further methods

Concrete class

Our concrete editor input class is implemented with the extended interface, to provide the additional methods for getting and saving the editor model. But the editor input doesn't hold these dates itself. He gets and passes them to another class responsible for its management. In our case this was a parser which transforms the editor model to a general domain model used in the remaining application. This was necessary, because the model of the graphical editor must hold additional data about the displayed elements, like the look and the size.

Compare methods

Additionally the methods `equals` and `hashCode` have to be implemented, because Eclipse checks the already opened editors if their input is equals to the new one which should be displayed. If this check returns true no additional editor view will be opened. That's the reason why we should override these two methods.

Listing 25 below shows an example implementation of these methods.

```
@Override
public boolean equals(Object obj) {
  boolean isEquals = false;

  if (obj instanceof PersonLocatorGraphEditorInput) {
    isEquals = equals((PersonLocatorGraphEditorInput) obj);
  }

  return isEquals;
}

public boolean equals(PersonLocatorGraphEditorInput editorInput)
{
  return getName().equals(editorInput.getName());
}

@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((fParser == null) ? 0 :
                                        fParser.hashCode());
  result = prime * result + ((name == null) ? 0 :
                                        name.hashCode());
  return result;
}
```

Listing 25: The `equals` and `hashCode` methods in the editor input class

Image

A further method which must be implemented is `getImageDescriptor`. According to the Javadoc and further descriptions *[IEditorInput08]*, the method should not return a null value to avoid problems. Therefore we followed this advice and returned `ImageDescriptor`s containing images. But to provide every time an image for that purpose may be cumbersome.

Therefore the class `ImageDescriptor` provides the factory method `getMis-`

`singImageDescriptor` to create an empty default image. It is a fast and correct way to implement the editor input class following the advices and recommendations of the Eclipse developer.

The code snippet shown in **Listing 26** is an example of such an implementation.

```
public ImageDescriptor getImageDescriptor() {
  return ImageDescriptor.getMissingImageDescriptor();
}
```

Listing 26: The `getImageDescriptor` method with a default image in the editor input class

Adapter

To allow an easy extension of our editor input class in a later step, without changing the code, we additionally implemented the method `getAdapter`. Cause of the interface `IEditorInput`, which extends `IAdaptable`, this method must be overridden anyway.

Because we have no adaption we could return in this method at its development time, we simple added the invocation of the `getAdapter` method of the `Platform`'s `AdapterManager`. This is an instruction widely used in the whole Eclipse RCP code base, why we decided to do so.

The code snippet in **Listing 27** shows such an implementation of the `getAdapter` method.

```
@SuppressWarnings("unchecked")
public Object getAdapter(Class adapter) {
  return Platform.getAdapterManager().getAdapter(this, adapter);
}
```

Listing 27: The `getAdapter` method in the editor input class

6.3.8.3.2 Problems

getEditorInput data

During the development of our graphical editor we implemented also an outline view, see chapter 6.3.8.8.1, showing the encapsulation of the shown elements amongst themselves: For example, which element is contained in another and so on. To provide this visualization, the outline needs also the classes shown in graphical editor. For an easier implementation, we read the data directly from the editor input. This worked well, but when we selected an element in the graphical editor, the corresponding entry in the outline view was not selected automatically by the system and vice versa.

This problem occurred, because the graphical editor and the outline view used the same data for the visualization, but not the same objects. This happened, because the editor input did not hold the objects but created them every time they were loaded. Therefore the system was not able to select the corresponding entry in the other view. We could solve this problem through the usage of the same model objects for both views.

The graphical editor was then responsible to hold and provide the diagram objects through its `getAdapter` method. This allowed us to use the same objects for the graphical editor as well as the outline and to minimize the dependencies

between these classes.

6.3.8.4 Diagram Elements

Overview

The most important part of the graphical editor after the editor class itself, are the elements which should be displayed. The visualization of an element in the editor is solved through an MVC-like *[MVC07]* class structure. This means a single element consists of a **model** class containing the required data, a **view** class for its visualization and a **controller** connecting them and handling the manipulation requests.

For this purpose GEF provides the abstract class `AbstractGraphicalEditPart` or derivations of it, which must be used for the implementation of the controllers. Also for the views, GEF offers a wide range of classes implementing the `IFigure` interface. Only for the model GEF makes no prescriptions.

Class structure

The **Figure 13** below shows the required class structure for the GEF visualization and manipulation.

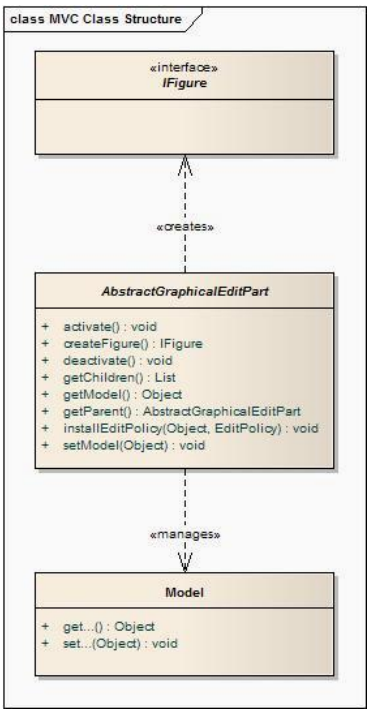

Figure 13: Schematic class diagram for the visualization of the editor elements

6.3.8.4.1 Model

Overview

The diagram model covers all the classes representing a visible element. They contain on the one hand the conventional information of a domain model and on the other hand the data used for the visualization like the size of the element or its location. This includes also classes for connections, because they are also

elements which must contain at least the data for the visualization.

In the example diagram shown in **Figure 14**, the representation in the model would consist of four classes. One for the *Person* and *Location*, another one for the *Category*, an additional for the *Dictionary* and the last one for the *connections* between them.

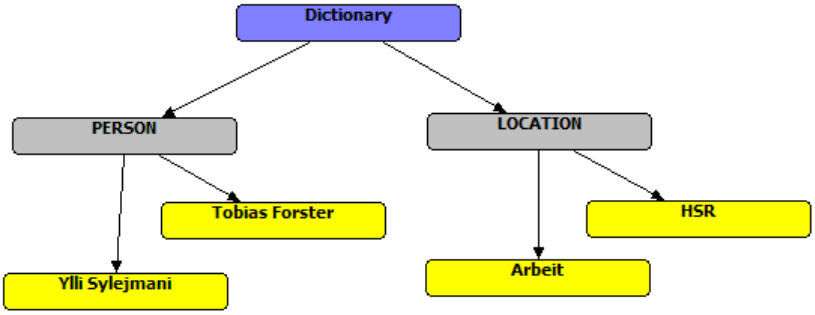

Figure 14: GEF diagram example

GEF does not make prescriptions about the shape of the model itself, only about its existence, because it creates the edit part, the controller out of the MVC-Pattern, based on them. Therefore you as the developer have the full choice how you implement it in detail.

6.3.8.4.1.1 Our Solution

Implementation

In our solutions, we decided to implement the diagram model manually like a conventional domain model and to extend it with additional attributes and classes to provide the full representation of its visualization. Instead of the manual implementation, a generation with EMF would also be possible. We decided against this possibility because the generated classes have strong dependencies to EMF classes. Additionally, EMF consists of about 32 plug-ins. This would increase the dependencies from our plug-in to them immensely and this for a simple diagram model.

Diagram model

Like mentioned above, we extended the diagram model with additional attributes and classes to provide its full visual representation. Additionally we had to add a new root class representing the whole diagram. This is necessary because the elements shown on the diagram must be contained somewhere, otherwise they would disappear after its creation. Normally we named this class *Diagram* or something like that.

The code snippet in **Listing 28** shows a class which is part of a diagram model. It represents a *Statement*.

```
public class GStatement extends Node {

  private long id = -1;

  public GStatement() {
    super();
    this.id = System.currentTimeMillis();
```

```
  }

  public GStatement(long id, String name, Rectangle layout) {
    super(name, layout);
    this.id = id;
  }

  @Override
  public boolean acceptAdditionalSourceConnections() {
    if (sourceConnections.size() >= 2) {
      return false;
    }

    return true;
  }

  @Override
  public boolean acceptAdditionalTargetConnections() {
    return false;
  }

  @Override
  public void addSourceConnection(Connection connection) {
    if (acceptAdditionalSourceConnections()) {
      super.addSourceConnection(connection);
    } else {
      throw new RuntimeException();
    }
  }

  @Override
  public void addTargetConnection(Connection connection) {
    throw new RuntimeException();
  }

  public Color getBackgroundColor() {
    return ColorConstants.darkBlue;
  }

  public Color getForegroundColor() {
    return ColorConstants.white;
  }

  public long getId() {
    return id;
  }
}
```

Listing 28: Diagram model class representing a simple statement

Node class

You may notice this class contains no attributes describing its visualization, like the size or the location. We located them in a general super class called `Node`. The reason for this decision is the fact that all classes visualized as nodes, can use them and don't have to implement the same code several times.

We also added further attributes with the corresponding methods to this class like listed below:

- The children attribute contains nodes which are contained in this node. In the graphical representation, they will be displayed inside this node.
- The source and target connections holding the `Connection` elements representing the links between two nodes. The reason for the differen-

tiation of the connections lies in the `EditPart` implementation (see chapter 6.3.8.4.3) and will be explained later.

Listing 29 shows an example for such a node class with all the required attributes and methods.

```
public abstract class Node implements PropertyChangeEvents {

  protected String name = "<<...>>";
  protected Rectangle layout = new Rectangle();
  protected List<Node> children = new ArrayList<Node>();
  protected List<Connection> sourceConnections =
                                new ArrayList<Connection>();
  protected List<Connection> targetConnections =
                                new ArrayList<Connection>();

  protected PropertyChangeSupport listeners;

  public static final int DEFAULT_HEIGHT = 25;
  public static final int DEFAULT_WIDTH = 150;

  public Node() {
    listeners = new PropertyChangeSupport(this);
  }

  public Node(String name) {
    this();
    this.name = name;
  }

  public Node(String name, Rectangle layout) {
    this(name);
    this.layout = layout;
  }

  public boolean acceptAdditionalSourceConnections() {
    return true;
  }

  public boolean acceptAdditionalTargetConnections() {
    return true;
  }

  public void addChild(Node child) {
    if (children.add(child)) {
      listeners.firePropertyChange(PROPERTY_NODE_ADD, null,
          child);
    }
  }

  public void addPropertyChangeListener(PropertyChangeListener
listener) {
    listeners.addPropertyChangeListener(listener);
  }

  public void addSourceConnection(Connection connection) {
    sourceConnections.add(connection);
    listeners.firePropertyChange(
        PROPERTY_NODE_CONNECTION_CHANGED, null, connection);
  }

  public void addTargetConnection(Connection connection) {
    targetConnections.add(connection);
    listeners.firePropertyChange(
```

```
        PROPERTY_NODE_CONNECTION_CHANGED, null, connection);
  }

  public boolean contains(Node child) {
    return children.contains(child);
  }

  public List<Node> getChildren() {
    return children;
  }

  public Rectangle getLayout() {
    return layout;
  }

  public String getName() {
    return name;
  }

  public List<Connection> getSourceConnections() {
    return new ArrayList<Connection>(sourceConnections);
  }

  public List<Connection> getTargetConnections() {
    return new ArrayList<Connection>(targetConnections);
  }

  public void removeChild(Node child) {
    if (children.remove(child)) {
      listeners.firePropertyChange(PROPERTY_NODE_REMOVE, child,
          null);
    }
  }

  public void removePropertyChangeListener(PropertyChangeListener
listener) {
    listeners.removePropertyChangeListener(listener);
  }

  public void removeSourceConnection(Connection connection) {
    sourceConnections.remove(connection);
    listeners.firePropertyChange(
        PROPERTY_NODE_CONNECTION_CHANGED, connection, null);
  }

  public void removeTargetConnection(Connection connection) {
    targetConnections.remove(connection);
    listeners.firePropertyChange(
        PROPERTY_NODE_CONNECTION_CHANGED, connection, null);
  }

  public void setLayout(Rectangle newLayout) {
    Rectangle oldLayout = layout;
    this.layout = newLayout;
    listeners.firePropertyChange(PROPERTY_NODE_LAYOUT,
                                 oldLayout, newLayout);
  }

  public void setName(String newName) {
    String oldName = name;
    this.name = newName;
    listeners.firePropertyChange(PROPERTY_NODE_NAME, oldName,
        newName);
  }
}
```

Listing 29: Diagram model base class representing a Node including notifications

Connection support

You may notice the class in **Listing 29** has two list attributes containing object from the type `Connection`. Like mentioned above, to show and manipulate connections between diagram elements, you need a corresponding model class. Otherwise to show connections, you must implement the `EditPart`s in a special way.

We decided against such a solution and implemented a class representing the connection. This also has the advantage to extend it in an easy way if it is required in the future, for example to describe them with a text.

The class itself is very simple, compared with those representing nodes in the diagram. It holds a reference to the source and target objects, in the example shown in **Listing 30** `Node`s, and a boolean value describing if they are connected or not.

```
public class Connection {

  private Node source = null;
  private Node target = null;

  private boolean isConnected;

  public Connection(Node source, Node target) {
    reconnect(source, target);
  }

  public void disconnect() {
    if (isConnected) {
      source.removeSourceConnection(this);
      target.removeTargetConnection(this);
      isConnected = false;
    }
  }

  public Node getSource() {
    return source;
  }

  public Node getTarget() {
    return target;
  }

  public void reconnect() {
    if (!isConnected) {
      source.addSourceConnection(this);
      target.addTargetConnection(this);
      isConnected = true;
    }
  }

  public void reconnect(Node newSource, Node newTarget) {
    if (newSource == null || newTarget == null ||
        newSource == newTarget) {
      throw new IllegalArgumentException();
    }

    disconnect();
    this.source = newSource;
```

```
    this.target = newTarget;
    reconnect();
  }
}
```

Listing 30: Diagram model base class representing a connection

When creating connections, new instances can be added easily to the `Node` class shown in **Listing 29** through the provided methods.

Notifications

When adding new elements to the diagram, or updating the content of them, the graphical editor must be notified about these changes.

For such a notification several Java technologies exists, each with its individual advantages and disadvantages. From the ones described in chapter 4.3.2, we chose the best solutions and analyzed them here, to find an optimal solution for our graphical editor.

- *Java's Observer implementation:* It's a simple and flexible implementation of notifications. But in the classes which want to notify registered listeners, it's required to inherit from `Observable`. Although this can be done by base classes like Node, it is restrictive.
- *JavaBeans property change:* Another easy implementation of a notification algorithm is the property change out of package `java.beans`. It's optimized for the use in java beans. Since the diagram model classes are also simple java beans, this implementation would fit perfectly. Additionally its use is very simple, like Java's Observer implementation, but without the restriction through the inheritance.
- *Eclipse's Data Binding:* A further alternative is Eclipse's Data Binding. Its implementation is designed to handle the synchronization of data between a user interface and the corresponding model classes. From the basic concept this would fit with GEF's MVC structure. But its implementation is large and inflexible, because it maps an attribute of the model directly with one of the view. Additionally it would increase the dependencies between the different classes in the MVC structure.

Because of its flexible structure and simple implementation, we decided to use the *JavaBeans property change*. It fits best with its possibilities and restrictions in the given GEF class structure. Therefore we used this observer implementation for our diagram model, which can be seen in **Listing 29**.

6.3.8.4.1.2 Problems

Differences between domain and diagram model

A difficult problem at the beginning of the graphical editor implementation was the structure of the diagram model. It is largely different compared with a conventional domain model, because it must consider the characteristics and restrictions of GEF.

In the following **Figure 15** the differences between a conventional domain model and the corresponding diagram model are shown through an example. The diagram model requires an additional root element, in this case *Diagram*. Additionally the *Person* is no longer directly connected with *Address* and *Location*, because they should be display as own nodes with a connection to *Person*. Otherwise they would be displayed as entries in the *Person* itself.

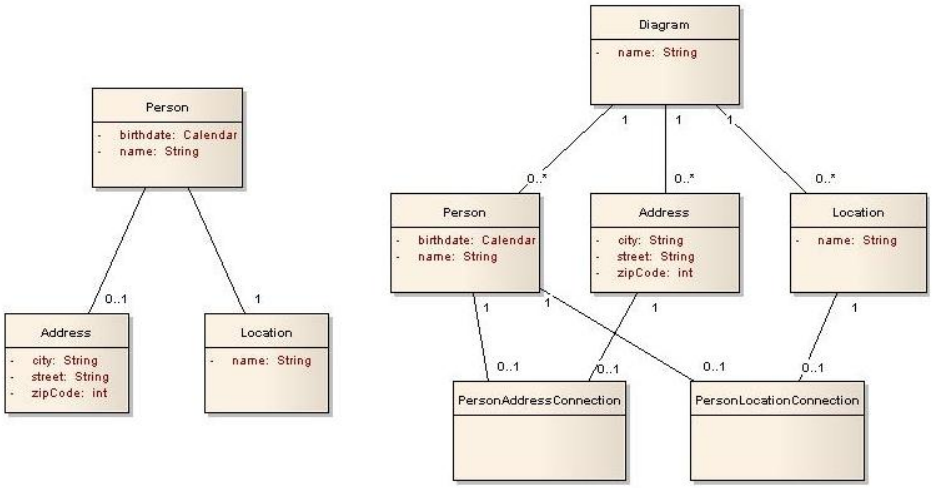

Figure 15: Conventional domain model (left) and the corresponding diagram model (right)

So this is the reason, why GEF and GMF needs an own model for its visualization. Otherwise they would be shown in a wrong way or the design and implementation would be messy.

New element disappears

A problem during the implementation of the diagram model was the fact, where to add new elements not yet contained or connected with an existing element.

Therefore we had to add a new root object representing the whole diagram which contains all the elements which should be shown in the diagram as nodes. After this extension the creation of new elements worked as expected.

6.3.8.4.2 Views

Overview

A further part of the MVC-based GEF class structure is the figure. It acts as the visible part, which is displayed in the graphical editor. It is created by the edit part which manages the figure. Through that the figure can focus on its core concerns, the visualization.

GEF provides by default a large amount of different figures *[Daum2008]* (classes implementing the interface `IFigure`), to provide an easier implementation. So it's not necessary to paint your figures on your own.

6.3.8.4.2.1 Our Solution

Implementation

We implemented our figures as own classes inheriting from `Figure` or a subclass of it and defined through an own interface. This allowed us to implement figures which can be exchanged easily through a substitute. But a disadvantage of this implementation style is the higher effort to create figures. **Listing 31** shows such an interface and an excerpt of the corresponding class.

```
public interface INodeFigure extends ITextFigure {
```

```
  public void setLayout(Rectangle layout);

  public void setBackgroundColor(Color background);

  public void setForegroundColor(Color foreground);
}

public class NodeFigure ... implements INodeFigure {
  ...
```

Listing 31: Definition of an interface for a figure

As an alternative, a figure can be created and configured in the `createFigure` method of the edit parts using the given GEF classes. This has the advantage of a smaller implementation effort, because no own classes must be developed, but on the other side you fuse the figure with the edit part. An exchange of it is only possible through a code change. Additionally the use of a figure in several edit parts is more complicated or even impossible. An example of such an implementation is shown in **Listing 36**.

Therefore we decided to implement own figures which can be reused, exchanged or extended in an easy way.

Layout

To extend a figure, you must add new elements like labels, lines or others. The way you can do this is very similar to Java Swing, where you set a `LayoutManager` and add afterwards the different elements to the container.

We had to define also a `LayoutManager` which organizes the elements contained in the figure. Afterwards, we could add other figures, like `Label`s, `Ellipse`s or others. In the same step the different subfigures and the figure itself can be configured and structured setting borders, colors, fonts and further more.

The following excerpt from a figure class shown in **Listing 32** uses a `RoundedRectangle` as the super class and adds a `Label` containing the name of the element it represents. In this case it's the name of a node.

```
public class NodeFigure extends RoundedRectangle
                         implements INodeFigure {

  private Label nameLabel = new Label();
  private ToolbarLayout layout = new ToolbarLayout();

  public NodeFigure() {
    layout.setMinorAlignment(ToolbarLayout.ALIGN_CENTER);
    setLayoutManager(layout);

    nameLabel.setFont(new FontRegistry().getBold("Verdana"));
    add(nameLabel);
  }

  public void setBackgroundColor(Color background) {
    super.setBackgroundColor(background != null ? background
        : ColorConstants.white);
  }

  public void setForegroundColor(Color foreground) {
    nameLabel.setForegroundColor(foreground != null ? foreground
        : ColorConstants.black);
  }
```

```
  public void setLayout(Rectangle layout) {
    getParent().setConstraint(this, layout);
  }

  public String getText() {
    return nameLabel.getText();
  }

  public Rectangle getTextBounds() {
    return nameLabel.getTextBounds();
  }

  public void setText(String text) {
    nameLabel.setText(text);
  }
}
```

Listing 32: Implementation of a figure using `RoundedRectangle` for the graphical editor

You can recognize the organization of the contained elements using the `ToolbarLayout`. Additionally you can see how to set the background color of the figure and the font type of the label.

Sub elements

Through the implementation of the method `getModelChildren` in the `EditPart` class, GEF will create edit parts for the child model classes returned in this method. The figures created by them, will be added to the figure of the parent. To reach this behavior we had nothing more to do, because this will be handled automatically by the graphical editor. The only thing we had to consider is to choose the appropriate `LayoutManager`; otherwise the sub elements may be displayed in unexpected manner.

Layout

GEF provides for this purpose a variety of different `LayoutManager`s. We often used the `ToolbarLayout`, the `XYLayout` and the `StackLayout`.

6.3.8.4.2.2 Problems

Sub elements

We had a cumbersome problem with the sub elements created through the edit parts of the model child elements. They were added to the parent figure in a wrong order. This means the child figures were added in a way they are shown above of the labels and lines we added in the parent figure itself.

We could solve this problem through a change of `LayoutManager`, which is responsible for the correct order of the elements in the parent figure.

6.3.8.4.3 Controller

Overview

The edit parts act in the given GEF class structure as the mediator between the model and the view (also known as *Figure*). In the MVC Pattern this type of class is called controller.

They are responsible to create a corresponding figure and to handle the requests on it. Additionally they must tell GEF if the figure has connections or children (embedded elements), which must be displayed. All the rest will be

handled by GEF or additional classes.

Typically, for each model instance a corresponding edit part instance exists, which handles the user interactions on the figure it creates. **Figure 16** shows such an object instance structure schematically.

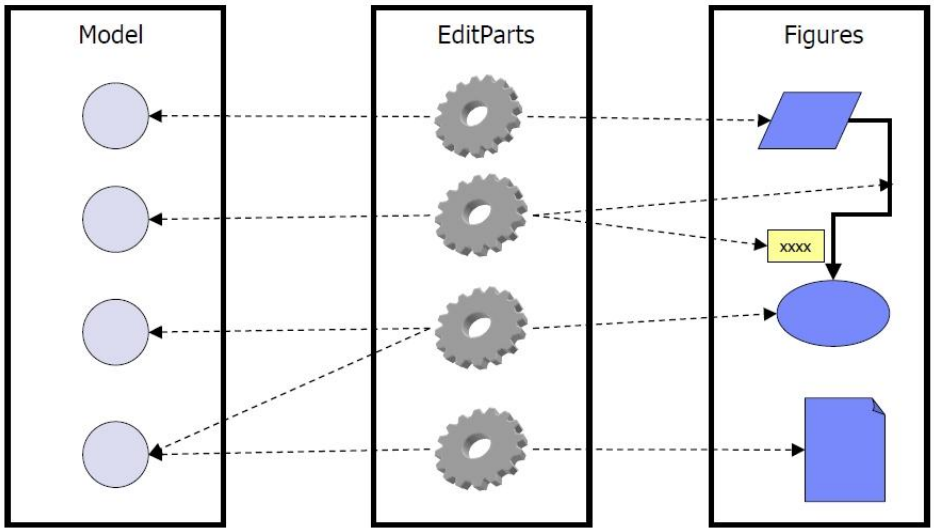

Figure 16: Model, EditPart and Figure instances to display a diagram (Source: *[Hudson04]*)

The edit parts will be created through a class implementing the `EditPartFactory` interface. The method `createEditPart` of this class will be called by GEF to get an edit part for a given instance of a model class. So the class has to decide which is the appropriate edit part and must create an instance of it.

Provided classes

GEF provides abstract classes which contains default implementations and hook-ins for the specific edit parts. Additionally GEF makes a distinction between edit parts of connections (`AbstractConnectionEditPart`) and those of normal graphical figures (`AbstractGraphicalEditPart`), like nodes. This allows an optimal support for them.

6.3.8.4.3.1 Our Solution

Implementation

In our solution of the graphical editor, we implemented for each model an own edit part class, to provide the specific behavior. For their implementation we used the `AbstractGraphicalEditPart` respectively the `AbstractConnectionEditPart` provided by GEF and overrode only the needed methods.

For classes with recurring methods or common configurations, we created abstract classes containing them. This allowed us to reduce duplicate code and to hold the specific classes compact.

Node

Our edit parts for the nodes override several methods. In a first step, we implemented the mandatory methods `createFigure` and `createEditPolicies`

to create the corresponding figure and to register edit policies to handle the interactions.

```
@Override
protected IFigure createFigure() {
  return new NodeFigure();
}
```

Listing 33: Example of the `createFigure` method in the PersonAttributeEditPart

Additionally we implemented the method `refreshVisuals`. This method will be called if the edit part is refreshed. The Javadoc proposes to use this it for refreshing the visuals. And for exactly this purpose we used it.

```
@Override
protected void refreshVisuals() {
  INodeFigure figure = (INodeFigure) getFigure();
  GPersonAttribute model = (GPersonAttribute) getModel();

  figure.setBackgroundColor(model.getBackgroundColor());
  figure.setForegroundColor(model.getForegroundColor());
  figure.setLayout(model.getLayout());
  figure.setText(model.getName());
  figure.repaint();
}
```

Listing 34: Example of the `refreshVisuals` method in the PersonAttributeEditPart

For edit parts whose model contains children, we additional overrode the method `getModelChildren`. Here we returned the children of the model. This force GEF to invoke for each object the `EditPartFactory` creating the whole edit part structure for the given model classes. This method must be overridden typically in the edit part representing the diagram, like **Listing 35** shows.

```
@Override
protected List<Node> getModelChildren() {
  return ((GStatementDiagram) getModel()).getChildren();
}
```

Listing 35: Example of the `getModelChildren` method in the DiagramEditPart

Connection

Really similar is the implementation of the edit parts for the connection. Instead the inheritance of `AbstractGraphicalEditPart`, it inherits from `AbstractConnectionEditPart`.

Also in this class we had to implement the mandatory methods `createFigure` and `createEditPolicies` in a first step. Different to the other edit parts is the method `createFigure`. We must return a `PolylineConnection` which can be created using the same method of the abstract super class. Before returning the connection, we made some configurations, as shown in **Listing 36**.

```
@Override
protected IFigure createFigure() {
  PolylineConnection figure =
              (PolylineConnection) super.createFigure();
```

```
  figure.setForegroundColor(ColorConstants.black);
  figure.setTargetDecoration(new PolygonDecoration());
  return figure;
}
```

Listing 36: Example of the `createFigure` method in the ConnectionEditPart

But this is only the first step. Afterwards we had to tell GEF which elements should be connected with connections. Therefore we had to do some extensions in those edit parts. First of all we implemented the interface `NodeEditPart` which forces us to override the method shown in **Listing 37**. Based to the shape of the figure, which was rectangular, we created new `ChopboxAnchor` elements and returned them.

```
public ConnectionAnchor getSourceConnectionAnchor(
        ConnectionEditPart connection) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getSourceConnectionAnchor(
        Request request) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
        ConnectionEditPart connection) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
        Request request) {
  return new ChopboxAnchor(getFigure());
}
```

Listing 37: Example of the anchor methods in the PersonAttributeEditPart

Additionally we had to tell GEF which connections start and end in this figure. We achieved this through overriding the methods `getModelTargetConnections` and `getModelSourceConnections`. The following **Listing 38** shows an example of them.

```
@Override
protected List<Connection> getModelTargetConnections() {
  GPersonAttribute model = (GPersonAttribute) getModel();
  return model.getTargetConnections();
}

@Override
protected List<Connection> getModelSourceConnections() {
  GPersonAttribute model = (GPersonAttribute) getModel();
  return model.getSourceConnections();
}
```

Listing 38: Example of the connection methods in the PersonAttributeEditPart

Notification

After interactions the data in the model changes, but the view must be updated as well. Therefore we had to register the edit part at the corresponding model to

get the notifications.

In a first step we implemented the interface `PropertyChangeListener` in our edit parts. It forced us to override the method `propertyChange` which receives the notifications. With the passed argument you can check the notification type and refresh or update the appropriate part of the view. **Listing 39** shows such a method.

```
public void propertyChange(PropertyChangeEvent evt) {
  if (evt.getPropertyName().equals(Node.PROPERTY_NODE_LAYOUT) ||
      evt.getPropertyName().equals(Node.PROPERTY_NODE_NAME)) {
    refreshVisuals();
  }
  if (evt.getPropertyName().equals(Diagram.PROPERTY_NODE_ADD) ||
      evt.getPropertyName().equals(Diagram.PROPERTY_NODE_REMOVE))
  {
    refreshChildren();
  }
  if (evt.getPropertyName().equals(
                  Node.PROPERTY_NODE_CONNECTION_CHANGED)) {
    refreshSourceConnections();
    refreshTargetConnections();
  }
}
```

Listing 39: Example of the `propertyChange` method in the PersonAttributeEditPart

To get the notifications we also had to implement the registration and deregistration from the model. The `AbstractGraphicalEditPart` provides for this purpose the methods `activate` and `deactivate`, like **Listing 40** shows.

```
@Override
public void activate() {
  super.activate();

  Node model = (Node) getModel();
  model.addPropertyChangeListener(this);
}

@Override
public void deactivate() {
  super.deactivate();

  Node model = (Node) getModel();
  model.removePropertyChangeListener(this);
    }
```

Listing 40: Example of the `activate` and `deactivate` methods in the PersonAttributeEditPart

Since we inherited all our models from the same base class, called `Node`, we were able to move the implementation of the methods `activate` and `deactivate` from the concrete edit parts in an abstract class. Thereby we could reduce duplicate code.

Factory

Like mentioned in the chapters above, the graphical editor requires a factory to create the edit parts for the given models. GEF provides for its implementation the interface `EditPartFactory`, which forces you to override the method

`createEditPart`. The method gets the parent edit part and the model for which a new edit part should be created.

Listing 41 below shows an example implementation of such an `EditPartFactory` creating edit parts for several different model class types.

```
public class PersonLocatorEditPartFactory implements EditPartFactory {

  public EditPart createEditPart(EditPart context, Object model)
  {
    EditPart part = null;

    if (model instanceof GStatementDiagram) {
      part = new DiagramEditPart();
    } else if (model instanceof GStatement) {
      part = new StatementEditPart();
    } else if (model instanceof Connection) {
      part = new ConnectionEditPart();
    } else if (model instanceof GPersonAttribute) {
      part = new PersonAttributeEditPart();
    } else if (model instanceof GLocationAttribute) {
      part = new LocationAttributeEditPart();
    } else {
      throw new RuntimeException("Model with type " +
          model.getClass() + " cannot be handled!");
    }

    part.setModel(model);
    return part;
  }
}
```

Listing 41: Example of the `createEditPart` method in a EditPartFactory

To allow the graphical editor the usage of this factory, it must be registered, before the content will be set. Like described in chapter 6.3.8.2, the method `configureGraphicalViewer` will be called before setting the content. Therefore we registered the factory on the editor in this method, like the following **Listing 42** shows.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  // EditPartFactory registration
  viewer.setEditPartFactory(new PersonLocatorEditPartFactory());

  ...
}
```

Listing 42: Registration of the `EditPartFactory` on the graphical editor

6.3.8.5 Commands

Overview

To apply changes on the diagram and its elements, GEF provides a *Command Processor [Buschmann1996]* infrastructure. Because of that the interactions must be encapsulated in command objects, which can be executed in a different time context. Through this encapsulation, the manipulation can also be undone if needed. As a further advantage of such an infrastructure, the application of the different changes can be done in a common way. This is possible through the use of the abstract class `Command` for the implementation of the command objects. It is not the same class like the undo mechanism of RCP uses.

Generally the command objects will be created and filled with data through the edit policies (see chapter 6.3.8.6), which handles the incoming requests from the user interface of the graphical editor.

Limitations

Because of these prescriptions, the possibilities to vary the implementation are very limited. Therefore our solution is in several ways very similar to the proposed implementation.

6.3.8.5.1 Our Solution

Implementation

In our solution we followed the suggestions of the GEF developer. For the implementation of the command objects, we used the abstract class `Command`.

For the application of the manipulation on the target object, we overrode the method `execute` and implemented it with the appropriate action. If the execution of the command objects depended on conditions, we additional overrode the method `canExecute` to announce the graphical editor if the command can be executed or not.

We did the same also for the `undo` and `canUndo` method to restore the initial state.

By default the `redo` method will invoke the `execute` method. In the command we implemented, we don't have to change or extend this behavior, because it was correct for us.

The **Listing 43** below shows an example of a `Command` object we typically implemented in our applications.

```
public class NodeCreateCommand extends Command {

  private GStatementDiagram parent = null;
  private Node newChild = null;

  @Override
  public boolean canExecute() {
    if (parent != null && newChild != null) {
      return true;
    } else {
      return false;
    }
  }

  @Override
  public boolean canUndo() {
```

```
    if (parent != null || newChild != null) {
      return parent.contains(newChild);
    } else {
      return false;
    }
  }

  @Override
  public void execute() {
    parent.addChild(newChild);
  }

  public void setLayout(Rectangle r) {
    if (newChild == null) {
      return;
    }
    newChild.setLayout(r);
  }

  public void setNewChild(Object newChild) {
    if (newChild instanceof Node) {
      this.newChild = (Node) newChild;
    }
  }

  public void setParent(Object parent) {
    if (parent instanceof GStatementDiagram) {
      this.parent = (GStatementDiagram) parent;
    }
  }

  @Override
  public void undo() {
    parent.removeChild(newChild);
  }
}
```

Listing 43: Example implementation of a command creating nodes

6.3.8.6 EditPolicies

Overview

To handle the user interactions on the user interface of the graphical editor and the visual objects, GEF provides a listener concept called EditPolicies. Their core purposes are the following ones:

- Creating command objects applying specific actions on the diagram elements.
- Feedback management for the graphical editor to show/erase the source and/or target feedback in response to request. For example if a connection between two elements can be created or not.

Compared with the typical listener structure, using an interface and register and deregister methods, the edit policies are more complex. Therefore they can handle more extensive user interactions covering several single interactions, like a mouse click or a key press.

GEF provides for the different user interactions abstract and default implementa-

tions which can be adapted easily. On top of this class hierarchy is the interface `EditPolicy`.

The registration of the edit policies on the elements, they should apply, is proposed to be made in the element's `createEditPolicies` method. Together with a constant, provided by the `EditPolicy` interface, an instance of the edit policy can be passed the method `installEditParts`.

6.3.8.6.1 Our Solution

Implementation

In our solution we implemented edit policies for different purposes, using specific abstract classes.

Requests on elements itself

To handle requests registered on the elements where the interaction should be applied to, we used the `ComponentEditPolicy`. An example here for is the handling of the deletion request, to create a delete command.

Such an example is shown in **Listing 44** below. It creates a delete command for the element the request is applied on.

```
public class NodeEditPolicy extends ComponentEditPolicy {

  @Override
  protected Command createDeleteCommand(
                                    GroupRequest deleteRequest) {
    DeleteCommand command = new DeleteCommand(
                    (Node) getHost().getModel(),
                    (Node) getHost().getParent().getModel());
    return command;
    }
}
```

Listing 44: Example implementation of an edit policy creating a delete command

Through the method `getHost` the model object belonging to the element where the request is applied on, can be gotten as well as other objects, like the parent model. They can be used for the command objects.

To apply the edit policy, it must be registered on the elements which a user may want to delete. The correct method to do this is the `createEditPolicies`, like the following code snippet in **Listing 45** shows.

```
@Override
protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE,
                    new NodeEditPolicy());
  ...
}
```

Listing 45: Registration of the edit policy allowing the deletion of an diagram element

Movement of diagram elements

Different to the interactions directly applied on the diagram elements, are those for manipulation their location and size. They will be handled by the edit policy of the parent element. For this purpose, GEF provides the abstract `XYLayoutEditPolicy`. It forced us to implement the methods `createChangeConstraintCommand` and `getCreateCommand`. The first one is invoked when the element is moved or resized, the other one when a new element is created.

Such an example is shown in **Listing 46** below. In the first method it creates a layout change command for the child of the element the request is applied on. The second parameter is the new location and size in form of a `Rectangle` element.

In the second method the new element which should be added, can be fetched from the passed request object like the location of the mouse click too.

```
public class NodeEditLayoutPolicy extends XYLayoutEditPolicy {

  @Override
  protected Command createChangeConstraintCommand(EditPart child,
        Object constraint) {

    Command command = null;

    if (child instanceof StatementEditPart ||
        child instanceof LocationAttributeEditPart ||
        child instanceof PersonAttributeEditPart ||
        child instanceof AttributeValueEditPart) {
      command = new NodeChangeLayoutCommand(
                                (Node) child.getModel(),
                                (Rectangle) constraint);
    }
    return command;
  }

  @Override
  protected Command getCreateCommand(CreateRequest request) {

    if (request.getType() == REQ_CREATE &&
        getHost() instanceof DiagramEditPart) {

      NodeCreateCommand command = new NodeCreateCommand();
      command.setNewChild(request.getNewObject());
      command.setParent(getHost().getModel());

      Rectangle constraint =(Rectangle)getConstraintFor(request);
      constraint.x = (constraint.x < 0) ? 0 : constraint.x;
      constraint.y = (constraint.y < 0) ? 0 : constraint.y;
      constraint.width = (constraint.width <= 0) ?
                  Node.DEFAULT_WIDTH : constraint.width;
      constraint.height = (constraint.height <= 0) ?
                  Node.DEFAULT_HEIGHT : constraint.height;

      command.setLayout(constraint);
      return command;
    }

    return null;
  }
}
```

Listing 46: Example implementation of an edit policy creating a create and move command

Like you have already noticed, this kind of edit policy must not be registered on the element which should be moved and resized, but on its parent where all this happens. We implemented the registration in the same way as the other ones, too with the method `insatllEditPolicy` like the following code snippet in **Listing 47** shows.

```
@Override
protected void createEditPolicies() {
  ...
  installEditPolicy(EditPolicy.LAYOUT_ROLE,
                    new NodeEditLayoutPolicy());
  ...
}
```

Listing 47: Registration of the edit policy allowing to move and resize the diagram elements

Connection handling

For creating and reconnecting connections between different elements, we also had to implement a different edit policy. For these interactions, GEF provides the abstract class `GraphicalNodeEditPolicy`, which forced us to implement several methods.

The two methods `getConnectionCreateCommand` and `getConnection-CompleteCommand` are used to create a new connection. The first method will be invoked when choosing the connection source and we create therefore a new command object. Consequently the second method will be called when the connection target was chosen and we added this element to the command, before it will be executed.

The following excerpt in **Listing 48** shows such an implementation.

```
public class ConnectionNodeEditPolicy extends GraphicalNodeEdit-
Policy {

  @Override
  protected Command getConnectionCompleteCommand(
        CreateConnectionRequest request) {

    ConnectionCreateCommand command = (ConnectionCreateCommand)
                                  request.getStartCommand();
    command.setTarget(getHost().getModel());
    return command;
  }

  @Override
  protected Command getConnectionCreateCommand(
        CreateConnectionRequest request) {

    ConnectionCreateCommand command =
                                   new ConnectionCreateCommand();
    command.setSource(getHost().getModel());

    request.setStartCommand(command);
    return command;
  }

  @Override
  protected Command getReconnectSourceCommand(
        ReconnectRequest request) {
```

```
    Connection connection = (Connection)
                    request.getConnectionEditPart().getModel();
    ConnectionReconnectCommand command =
                    new ConnectionReconnectCommand(connection);
    command.setNewSource(getHost().getModel());

    return command;
  }

  @Override
  protected Command getReconnectTargetCommand(
        ReconnectRequest request) {

    Connection connection = (Connection)
                    request.getConnectionEditPart().getModel();
    ConnectionReconnectCommand command =
                    new ConnectionReconnectCommand(connection);
    command.setNewTarget(getHost().getModel());

    return command;
  }
}
```

Listing 48: Example implementation of an edit policy creating a create and move command

The other two methods `getReconnectSourceCommand` and `getReconnectTargetCommand` are invoked when a connection is moved to a new source respectively target element. In both methods we got the connection through the passed request object. After collecting all required data and objects, we created a new command with them.

Like the edit policy for moving and resizing the elements too, we had to register this policy not in the connection elements, but in the elements which will be connected. We implemented the registration in the same way like the other ones too with the method `insatllEditPolicy` like the following code snippet in **Listing 49** shows.

```
@Override
protected void createEditPolicies() {
  ...
  installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
                    new ConnectionNodeEditPolicy());
  ...
}
```

Listing 49: Registration of the edit policy allowing to establish and reconnect connections

Further edit policies

For other manipulations we used further edit policies given by GEF, we don't have to implement or adjust. For example in the edit part of the diagram root element, we registered the `RootComponentEditPolicy` or in the edit part of the connection the `ConnectionEndpointEditPolicy`. The first one prevents the root element from its deletion. The second is responsible to place handles on the connection to drag them.

6.3.8.7 Direct Edit

Overview

Through the edit policies and the corresponding commands, we can only create, move and delete elements in the shown diagram. The change of the shown text in a direct manner is not possible, only through the implementation of wizards or something else.

For this purpose GEF provides the direct edit feature. It allows changing the text of an element through a simple click or double-click on it. Because of that, the implementation of wizards invoked through context menus is omitted or can be minimized.

GEF supports the implementation of such a feature through abstract and default implementations like the `DirectEditPolicy`, the `DirectEditManager` and the `CellEditorProvider`.

Structure

The diagram in **Figure 17** shows the classes involved in the direct edit feature schematically.

Beside the existing edit part and the view, the abstract class `DirectEditManager` is a central part. It creates the `CellEditor` as well as its locator. They control the direct edit and its visualization and have direct access to the figure.

To use the `DirectEditManager`, it must be instantiated and invoked from the edit part, because it's the only class which is able to pass the figure where the `CellEditor` should be applied on and knows the moment when to show the direct edit.

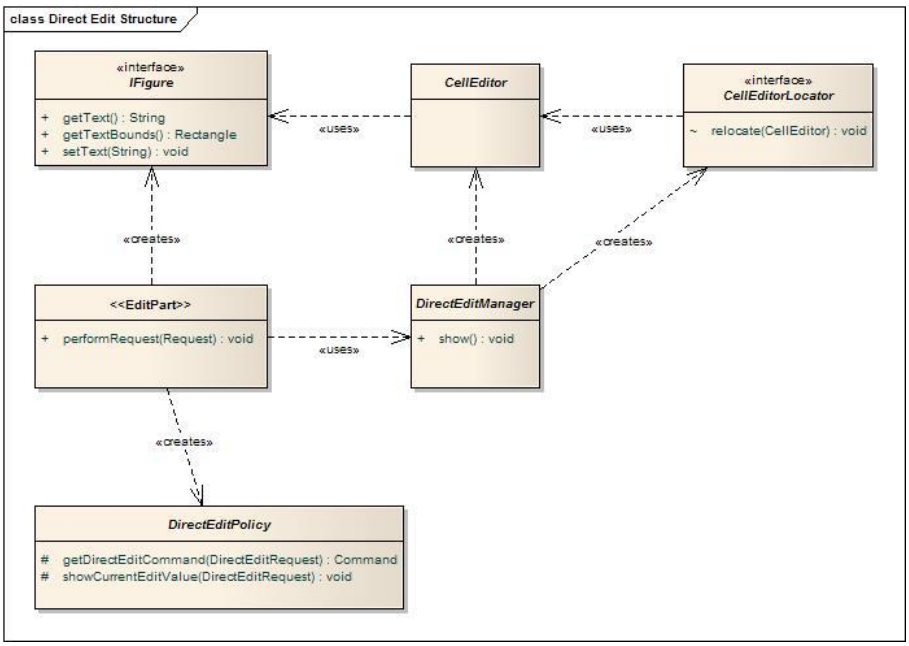

Figure 17: Schematic diagram of the classes involved in the direct edit

To handle the interactions on the figures and the corresponding edit parts afterwards, also special `DirectEditPolicy` is required. It is responsible to show the changed text as well as creating command objects to change it permanently.

6.3.8.7.1 Our Solution

Implementation

We also used the direct edit feature in our graphical editors displaying nodes with text. Additionally we tried to separate the common parts from the editor specific ones, to minimize code duplicity. The general parts we decided to locate in the same bundle like the other general parts of the graphical editor.

Figure

To those belongs the interface `ITextFigure`. We defined it for all the figures which does not inherit from `Label` and therefore must provide own methods for accessing the text. This allows us to handle the different figures in a common manner. Otherwise we had to implement if statements in several cases to handle them, which is cumbersome and increases the dependencies unnecessary.

The **Listing 50** below shows our implementation of the `ITextFigure` interface forcing to implement three methods. Two of them are for accessing the text. The third one returns the bounds of the text, which consists of its location and size.

```
public interface ITextFigure extends IFigure {

  public String getText();
  public void setText(String text);
  public Rectangle getTextBounds();
}
```

Listing 50: Definition of the `ITextFigure` interface for a common handling of the figures

For the implementation of our concrete figures we used the `ITextFigure`, in cases where they contained a text which should be changed with the direct edit feature. The listings in chapter 6.3.8.4.2 show examples of such an implementation.

Edit part

Additionally to the figure itself, we also had to extend the edit part classes. This includes on the one hand the registration of an additional edit policy as well as overriding the method `performRequest`, like **Listing 51** shows.

```
public class PersonAttributeEditPart extends AbstractNodeEditPart
    implements NodeEditPart {

  protected DirectEditManager manager;

  @Override
  protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE,
        new NodeEditPolicy());
    installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
        new ConnectionNodeEditPolicy());
    installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE,
        new NodeDirectEditPolicy());
  }

  ...

  @Override
  public void performRequest(Request request) {

    if (request.getType() == RequestConstants.REQ_DIRECT_EDIT
        || request.getType() == RequestConstants.REQ_OPEN) {
```

```
      if (manager == null) {
        manager = new TextDirectEditManager(this,
            (INodeFigure) getFigure());
      }

      manager.show();
    }
  }

  ...
}
```

Listing 51: Extension of the an edit part for the direct edit

Edit policy

The edit policy we had to implement and install in the edit part must inherit from the abstract class `DirectEditPolicy`. It forced us to implement the two methods `getDirectEditCommand` and `showCurrentEditValue`.

We decided to split the implementation of these methods in two classes, an abstract super class and a specific one. In the super class we located the common parts which cover mainly the `showCurrentEditValue` method. In this method we only had to get the value out of the request's `CellEditor` and to set it in the figure. Cause of the implementation of the `ITextFigure` interface, we could reduce the possible classes on two types, like shown in **Listing 52**.

```
public abstract class AbstractDirectEditPolicy
    extends DirectEditPolicy {

  @Override
  protected void showCurrentEditValue(DirectEditRequest request)
  {
    String text = (String) request.getCellEditor().getValue();

    if (getHostFigure() instanceof Label) {
      ((Label) getHostFigure()).setText(text);
    } else if (getHostFigure() instanceof ITextFigure) {
      ((ITextFigure) getHostFigure()).setText(text);
    }
  }
}
```

Listing 52: `AbstractDirectEditPolicy` with the common parts of the edit policy

The concrete parts are located in the specific class inherited from the abstract one. In this method we only had to implement the method `getDirectEditCommand`, which creates and returns a command for changing the text of the given model element. This method is implemented in the same style as the most other edit policies, like **Listing 53** shows.

```
public class NodeDirectEditPolicy
    extends AbstractDirectEditPolicy {

  @Override
  protected Command getDirectEditCommand(
      DirectEditRequest request) {

    return new NodeRenameCommand((Node) getHost().getModel(),
        (String) request.getCellEditor().getValue());
  }
}
```

Listing 53: Edit policy with the project specific implementations for the direct edit

The command used in this method is implemented in the same style as those already described in chapter 6.3.8.4.3.

DirectEdit-Manager

Different is the method `performRequest` in the edit part. It instantiates and shows a `DirectEditManager` if a direct edit request (single click on the figure) or an open request (double-click on the figure) is received.

Its only responsibility is to create and initialize the `CellEditor` and its `CellEditorLocator`, which visualizes and handles the direct edit.

We decided to implement the `DirectEditManager` in a common manner to allow its reusability and to locate it in the bundle with the general parts. This was especially possible through the `ITextFigure` interface which helps to restrict the possible figure types. In our solution we also extended our `TextDirectEditManager` with a `TextVerifyListener`. He calculates the optimal size of the shown edit field after each keyboard entry. This allows a better visualization of the changes, because it increases respectively decreases the size of the field steadily. Otherwise they would not be displayed properly.

CellEditor

The `CellEditor` itself is fully provided by GEF, which supports the developer with different implementations of the direct edit. It allows therefore not only the direct edit through text entry, but also using checkboxes, combo boxes or dialogs.

The definition and instantiation of the used type is very simple, because only the class has to be passed to the constructor of the abstract super class `DirectEditManager`. The rest will be handled by this class.

```
public TextDirectEditManager(GraphicalEditPart source,
        ITextFigure figure) {
  super(source, TextCellEditor.class,
      new TextCellEditorLocator(figure));

  this.textFigure = figure;
}
```

Listing 54: Definition of the `TextCellEditor` constructor of the `DirectEditManager`

CellEditor-Locator

A little different is the `CellEditorLocator`, which is responsible to locate the field for the direct edit on the right position. For its implementation GEF provides the interface `CellEditorLocator`. The only method we had to implement is `relocate`. The passed argument `CellEditor` allows an easy access to the figure we are editing.

Here we used now the return value of the figure method `getTextBounds` to locate the edit field properly, like the excerpt in **Listing 55** shows.

```
public class TextCellEditorLocator implements CellEditorLocator {

  private ITextFigure textFigure;
  private Label labelFigure;
```

```
  public TextCellEditorLocator(Label figure) {
    this.labelFigure = figure;
  }

  public TextCellEditorLocator(ITextFigure figure) {
    this.textFigure = figure;
  }

  public void relocate(CellEditor celleditor) {
    Text textControl = (Text) celleditor.getControl();
    Point preferredSize = textControl.computeSize(SWT.DEFAULT,
        SWT.DEFAULT);

    Rectangle textBounds = null;

    if (labelFigure != null) {
      textBounds = labelFigure.getTextBounds();
    } else {
      textBounds = textFigure.getTextBounds();
    }

    textControl.setBounds(textBounds.x - 1, textBounds.y - 1,
        preferredSize.x + 1, preferredSize.y + 1);
  }
}
```

Listing 55: A `CellEditorLocator` implementation for locating a text entry field

Through the use of the `ITextFigure`, we could also implement this class in a very generic manner. Therefore we located it in the bundle with the common parts of the graphical editor too.

6.3.8.7.2 Problems

DirectEditManager without TextVerifyListener

In a first step we implemented our `DirectEditManager`, creating the `CellEditor` for the text entry without the `TextVerifyListener`. The solution worked fine, but the visualization was not optimal, because some of the changes were hidden. The cause was the size of the edit field, which didn't increase in the same way we entered new text.

So we decided to implement the `TextVerifyListener`, like described in *[Ermel08-03]*, to provide the user a better visualization of the changed text.

6.3.8.8 Palette

Overview

The default way to create new diagram elements is through the tools provided by the palette. The palette itself is a special toolbar, typically shown on the left or right side of the editor.

GEF provides on the one hand two special abstract classes for the editor implementation, which integrates the palette automatically. This can be either `GraphicalEditorWithPalette` showing a simple palette or the `GraphicalEditorWithFlyoutPalette` with a palette which can be moved and minimized.

Additionally, abstract and default implementations for the palette tools are also provided which can be used to add tools. Otherwise you have an empty palette which provides no functions.

6.3.8.8.1 Our Solution

Editor

For our implementation of our graphical editors, we used the `GraphicalEditorWithFlyoutPalette` instead the `GraphicalEditor`. The change of the super class forced us to override additionally the method `getPaletteRoot`. Its return value is the root element for the palette viewer.

We chose this class because it has an advanced palette allowing its movement and minimization. So the user of the graphical editor can decide himself if he wants to see the palette or not and on which side.

Palette creation

Although the method `getPaletteRoot` must be overridden in the editor class, we decided to do the actual creation of the palette with its tools in an own factory class. The reason for this decision is to focus the two classes on their key concerns. Additionally such a solution allows also exchanging the palette factory easily through a substitute. In combination with a wiring through Spring, this provides a very flexible implementation.

The following **Listing 56** shows our typical implementation of the `getPaletteRoot` method simply invoking the palette factory class.

```
@Override
protected PaletteRoot getPaletteRoot() {
  return paletteFactory.createPalette();
}
```

Listing 56: Implementation of the `getPaletteRoot` method in the graphical editor

For the implementation of the palette factory class we defined a very simple interface which provides the base for a flexible configuration and exchange of the concrete classes. The **Listing 57** below shows our interface defining only the method `createPaletteRoot`.

```
public interface PaletteFactory {

  public PaletteRoot createPaletteRoot();
}
```

Listing 57: Definition of the `PaletteFactory` interface for concrete implementations

Palette tools

The concrete implementation of the `PaletteFactory` interface has to create the `PaletteRoot` and consequently the whole palette with its tools.

In the main method of the palette factory class, we created a new `PaletteRoot`, which contains the remaining parts of the palette, like groups and tools. Further we decided to group the tools depending on the function they provide in own methods. So we implemented for example a group/method for the general tools like the selection or marquee and one for the nodes we want to create, like

the **Listing 58** shows.

In these methods we created new classes inherited from `PaletteContainer` representing groups. We also created specific tool entries and added them to the palette container. GEF provides several classes for grouping tool entries distinguishing only in its graphical representation.

```
public class PersonLocatorPaletteFactory implements PaletteFacto-
ry {

  public PaletteRoot createPaletteRoot() {
    PaletteRoot palette = new PaletteRoot();

    palette.add(createToolsGroup(palette));
    palette.add(createNodesGroup());

    return palette;
  }

  private PaletteEntry createNodesGroup() {
    PaletteDrawer toolbar = new PaletteDrawer("Nodes");

    toolbar.add(new CreationToolEntry("Statement",
        "Create a new statement",
        new NodeCreationFactory(GStatement.class), null, null));
    toolbar.add(new CreationToolEntry("Person",
        "Create a new person",
        new NodeCreationFactory(GPersonAttribute.class), null,
        null));
    toolbar.add(new CreationToolEntry("Location",
        "Create a new location",
        new NodeCreationFactory(GLocationAttribute.class), null,
        null));
    toolbar.add(new CreationToolEntry("Attribute value",
        "Create a new attribute value",
        new NodeCreationFactory(GAttributeValue.class), null,
        null));

    return toolbar;
  }

  private PaletteEntry createToolsGroup(PaletteRoot palette) {
    PaletteToolbar toolbar = new PaletteToolbar("Tools");

    // Selection tool
    ToolEntry selectionToolEntry =
        new PanningSelectionToolEntry();
    toolbar.add(selectionToolEntry);
    palette.setDefaultEntry(selectionToolEntry);

    // Marquee tool
    toolbar.add(new MarqueeToolEntry());

    // Connection tool
    toolbar.add(new ConnectionCreationToolEntry("Connection",
        "Create a new connection", new EmptyCreationFactory(),
        PersonLocatorGraphPlugin
            .getImageDescriptor("icons/connection_s16.gif"),
        PersonLocatorGraphPlugin
            .getImageDescriptor("icons/connection_s24.gif")));

    return toolbar;
  }
}
```

Listing 58: Implementation of a concrete palette factory using the `PaletteFactory` interface

Like you have already noticed in the listing above, we created the tool entries in different ways. The easiest ones are the selection and marquee tool, because they are already provided by GEF. So we could use the default implementations.

Different are the ones for the creating nodes and connections. The constructors of these tool entries are much longer. They consist of the *visible text*, the *tooltip text*, a *creation factory*, a *small* and a *large icon*.

Creation Factory

The creation factory is used in the conventional manner through a selection of the entry and the subsequent creation through a click in the diagram. Its purpose is to create and return a new instance of the model class. The addition to the existing elements is handled through edit policies already described in previous chapter.

For the implementation of the creation factories, GEF provides the interface `CreationFactory` forcing to implement the two methods `getNewObject` and `getObjectType`. The first method returns a new instance of the object and the second one return the type of the object created in the first method.

In our solution we decided to implement a general `CreationFactory` creating the new model instances through Java Reflexion. So we had to pass only the class type. The reason for this decision is on the one hand the simplicity of the implementation. We don't have to implement a factory for each model element, and on the other hand the flexibility we gain through this solution.

The **Listing 59** shows the implementation of such a general `CreationFactory` using Java Reflexion.

```
public class NodeCreationFactory implements CreationFactory {

  private Class<? extends Node> type;

  public NodeCreationFactory(Class<? extends Node> type) {
    this.type = type;
  }

  public Object getNewObject() {
    Object obj = null;

    if (type != null) {
      try {
        obj = type.newInstance();
      } catch (Exception exc) {
        // do nothing
      }
    }

    return obj;
  }

  public Object getObjectType() {
    return type;
  }
}
```

Listing 59: Implementation of flexible and general `CreationFactory`

Edit Policies & Commands

Like mentioned above, the handling of the interaction on the diagram will be done by the edit policies, which creates corresponding `Command` objects. So we had to extend them.

To create nodes, we must override and implement the method `getCreateCommand` in our edit policy inherited from `XYLayoutEditPolicy`. From the passed `CreateRequest`, we got the new object using the `getNewObject` method and its parent, where it should be added to. After calculating its location and size, we could create and return the command object. **Listing 60** below shows such a method.

```
@Override
protected Command getCreateCommand(CreateRequest request) {
  if (request.getType() == REQ_CREATE &&
      getHost() instanceof DiagramEditPart) {

    NodeCreateCommand command = new NodeCreateCommand();
    command.setNewChild(request.getNewObject());
    command.setParent(getHost().getModel());

    Rectangle constraint = (Rectangle) getConstraintFor(request);
    constraint.x = (constraint.x < 0) ? 0 : constraint.x;
    constraint.y = (constraint.y < 0) ? 0 : constraint.y;
    constraint.width = (constraint.width <= 0) ?
        Node.DEFAULT_WIDTH : constraint.width;
    constraint.height = (constraint.height <= 0) ?
        Node.DEFAULT_HEIGHT : constraint.height;

    command.setLayout(constraint);
    return command;
  }
  return null;
}
```

Listing 60: Implementation of a concrete palette factory using the `PaletteFactory` interface

The command object which the edit policy creates, adds the new created object simply to the parent. Through the notification in the model classes the corresponding `EditPart`s should be notified which handles the remaining steps.

6.3.8.9 Drag & Drop

Overview

An important feature for creating new diagram elements is the drag & drop. Also GEF provides the possibility to implement the drag & drop feature *[Bordeau03]*, which is very similar to the one used in SWT *[Irvine03]*. For example it allows to drag a tool from the palette to the graphical editor and to drop there as a new element. It's an alternative to the creation of new elements through selecting the tool in the palette and to create a new element through a click in the graphical editor.

The introduction of such functionality requires on the one hand changes in the palette and on the other hand in the graphical editor itself. GEF provides for this purpose interfaces and abstract implementations like the `AbstractTransferDropTargetListener`, but also more convenient ones like the `Template-`

`TransferDragSourceListener` or the `TemplateTransferDropTarget-Listener` and special tools for the palette.

6.3.8.9.1 Our Solution

Implementation

In our solution of the graphical editor using the drag & drop feature, we decided to follow mainly the advices of the GEF developers. Therefore we had to do changes in the palette factory as well as an extension of the graphical editor's configuration.

Palette

In our factory creating the palette, we used until now the `CreationToolEntry` to create new elements, which can be added afterwards to the diagram. We replaced this class with the `CombinedTemplateCreationEntry`, which requires a further parameter. Its purpose is to describe the template object, which will be used for the instantiation. **Listing 61** shows an excerpt out of our palette factory with the new tool entry class.

```
private PaletteEntry createNodesGroup() {
  PaletteDrawer toolbar = new PaletteDrawer("Nodes");

  toolbar.add(new CombinedTemplateCreationEntry("Statement",
      "Create a new statement", GStatement.class,
      new NodeCreationFactory(GStatement.class), null, null));
  toolbar.add(new CombinedTemplateCreationEntry("Person",
      "Create a new person", GPersonAttribute.class,
      new NodeCreationFactory(GPersonAttribute.class), null,
      null));
  toolbar.add(new CombinedTemplateCreationEntry("Location",
      "Create a new location", GLocationAttribute.class,
      new NodeCreationFactory(GLocationAttribute.class), null,
      null));
  toolbar.add(new CombinedTemplateCreationEntry(
      "Attribute value", "Create a new location",
      GAttributeValue.class,
      new NodeCreationFactory(GAttributeValue.class), null,
      null));

  return toolbar;
}
```

Listing 61: Implementation of the palette tools with the `CombinedTemplateCreationEntry`

What kind of object to pass, is in the choice of the developer. We decided to pass the class meta object type, because it allows the instantiation of a new object through Java Reflexion. Additionally this is a very flexible solution for extensions in the future.

As a single restriction we must provide a constructor without parameters, otherwise this instantiation would fail. But in our opinion this is only a small constraint compared with the advantages of such a solution.

Using other template objects as parameter would lead to a more difficult implementations of the subsequent handling.

Drop target

In a further step, we had to implement a `DropTargetListener` which han-

listener

dles the instantiation of a new object when the user drops the element in the diagram. GEF provides for a simplified implementation the class `TemplateTransferDropTargetListener`.

Through overriding the method `getFactory`, we could return an instance of a `CreationFactory` class, which is responsible for the instantiation of a new diagram element. For this purpose we used the `NodeCreationFactory` class, we already used in the palette, like shown in **Listing 62**.

The method's only parameter is the template object defined in the `CombinedTemplateCreationEntry`. Cause of the generic implementation of the `NodeCreationFactory`, we only had to pass this template object casted to a class. The rest will be handled by the factory and the edit policies in the edit parts of the graphical editor.

```
public class NodeTemplateTransferDropTargetListener extends
        TemplateTransferDropTargetListener {

  public NodeTemplateTransferDropTargetListener(EditPartViewer
viewer) {
    super(viewer);
  }

  @Override
  @SuppressWarnings("unchecked")
  protected CreationFactory getFactory(Object template) {
    return new NodeCreationFactory(
        (Class<? extends Node>) template);
  }
}
```

Listing 62: Implementation of the `TemplateTransferDropTargetListener`

Graphical editor

In the last step of our implementation we had to extend the graphical editor, because we must register the `DragTargetListener` and the `DropTargetListener` on the appropriate classes.

The `DragTargetListener` must be registered in the `PaletteViewer`, because the user drags the element from there. For this purpose we had to get it from the `EditDomain`, like shown in **Listing 63**.

The `DropTargetListener` on the other side must be registered on the `GraphicalViewer`, where the user drops the element form the palette. Otherwise nothing would happen.

```
@Override
protected void initializeGraphicalViewer() {
  super.initializeGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();
  dropTargetListener =
      new NodeTemplateTransferDropTargetListener(viewer);
  viewer.addDropTargetListener(dropTargetListener);

  PaletteViewer paletteViewer =
      getEditDomain().getPaletteViewer();
  dragTargetListener = new TemplateTransferDragSourceListener(
      paletteViewer);
```

```
  paletteViewer.addDragSourceListener(dragTargetListener);
}
```

Listing 63: Configuration and implementation of the drag & drop in the graphical editor

6.3.8.10 Outline

Overview

An alternative visualization of the elements in the graphical editor provides the outline view. It shows the elements in a tree structure considering especially its nesting.

GEF provides abstract and default classes which simplifies the implementation of this view. Some of them are very similar to those in the graphical editor itself, because the GEF developers implemented the outline based on the same concept like the elements in the graphical editor, using a MVC-like class structure. Therefore several edit parts, inherited from `AbstractTreeEditPart`, are needed to show the different entries in the tree and an `EditPartFactory` for its creation.

The outline itself must be implemented using the class `ContentOutlinePage`. Its initialization and configuration is similar to the one of the graphical editor. Additionally it must be linked with the graphical editor. Otherwise manipulation in one view will not be shown in the other.

6.3.8.10.1 Implementation styles

Intention

The implementation of the outline view can be done in two different ways. In both an additional class using `ContentOutlinePage` will be used. But in one case it is located as an inner class in the editor whereas in the other case the class is independent.

Normally the implementation of complex inner classes is not considered, because it's bad design through the high coupling. But in this case we balance this solution with the independent one, because the outline has very strong dependencies to the graphical editor, caused mainly through the synchronization of these views.

Alternatives

Alternative 1: Inner class in the graphical editor

A first solution for the implementation of the outline view is to locate the class as an inner class in the graphical editor. Thus the outline class can access all the methods and attributes of the graphical editor which is necessary for the synchronization between them.

This implementation style is widely used by the GEF developer in their example implementations.

Advantage	Disadvantage
• The outline view can access all the attributes and methods of the graphical editor, espe-	• Generally, it's bad design to extend a class with large inner classes, because they will be coupled strongly.

cially the protected ones. • Several attributes and methods, which are required by the outline view, are protected or private in the graphical editor. Therefore they must be made accessible from outside if the outline is located in an independent class. This can be omitted with an inner class solution. • This implementation style is used by the GEF developers in almost all of their examples and has proven as a suitable solution.	• The outline is automatically integrated in the graphical editor even if the developers don't want it, especially in the cases where an abstract class contains this implementation. • The outline view is implemented specifically for the graphical editor it is located in. Its reusability is not possible. Therefore it must be implemented for each graphical editor.

Alternative 2: Independent class outside the graphical editor

An alternative solution to the inner class is the implementation of the outline view in a separate class. Therefore it's independent from the specifics of the graphical editor, which allows a reuse of the same class for several graphical editors.

Advantage	Disadvantage
• Generally it's good design to locate different concerns in own classes. • The outline view can be implemented generally which allows its reusage for several graphical editors. • The developer can decide itself if an outline view should be shown or not. It is not automatically integrated in the abstract class. • The graphical editor class is more focused and therefore easy to extend, maintain and test.	• The graphical editor must provide access to protected resources required for the outline view.

Decision

Based on the comparison of the two alternative solutions, we decided to implement the outline view in a separate class, because the advantages overbalance. Although we must provide access to several attributes and methods from the graphical editor to be used in the outline view, it helps us to keep the design clean.

Additionally the implementation of a generic outline view is only possible in the separate class, because the inner class has automatically dependencies to the graphical editor. Therefore a separate implementation would be necessary for

each graphical editor.

6.3.8.10.2 Our Solution

Implementation

Based on the decision in the chapter above, we located our outline view in a separate class independent from the specific graphical editor. For its implementation we inherited from `ContentOutlinePage`, which provides an empty outline view.

Outline view

This forced us to implement a constructor and to pass an instance of a viewer to the constructor of the super class. We passed a new instance of `TreeViewer`, to show an outline with a tree, like known from Eclipse.

Additionally we added the `GraphicalEditor` as a parameter to our constructor, to get access to its attributes and methods. We decided to provide them through the method `getAdapter`. The reason is mainly because it is a very flexible solution, which allows an easy extension in the future. Additionally it is the preferred way in Eclipse RCP to provide additional resources form a class, without changing its interfaces through new methods and further more.

The **Listing 64** below shows amongst others an example of such a constructor. Note we pass a `GraphicalEditor`, because our outline view should be generic. Otherwise it would depend on a specific graphical editor class.

```
public class DefaultOutlinePage extends ExtContentOutlinePage {

  protected SashForm sash;
  protected GraphicalEditor editor;
  protected EditPartFactory editPartFactory = ...;

  public DefaultOutlinePage() {
    super(new TreeViewer());
  }

  @Override
  public void createControl(Composite parent) {
    sash = new SashForm(parent, SWT.VERTICAL);

    getViewer().createControl(sash);
    getViewer().setEditDomain(
        (EditDomain) getEditorResource(EditDomain.class));
    getViewer().setEditPartFactory(editPartFactory);
    updateContents();

    SelectionSynchronizer synchronizer =
        getEditorResource(SelectionSynchronizer.class);
    synchronizer.addViewer(getViewer());
  }

  @Override
  public void dispose() {
    SelectionSynchronizer synchronizer =
        getEditorResource(SelectionSynchronizer.class);
    synchronizer.removeViewer(getViewer());

    super.dispose();
  }
```

```
  @Override
  public Control getControl() {
    return sash;
  }

  @SuppressWarnings("unchecked")
  protected <T> T getEditorResource(Class<?> resourceType) {
    T resource = null;

    if (editor.getAdapter(resourceType) != null) {
      resource = (T) editor.getAdapter(resourceType);
    }

    return resource;
  }

  @Override
  public void init(IPageSite pageSite) {
    super.init(pageSite);

    IActionBars bars = getSite().getActionBars();
    ActionRegistry registry =
        getEditorResource(ActionRegistry.class);

    bars.setGlobalActionHandler(ActionFactory.UNDO.getId(),
        registry.getAction(ActionFactory.UNDO.getId()));
    bars.setGlobalActionHandler(ActionFactory.REDO.getId(),
        registry.getAction(ActionFactory.REDO.getId()));
    bars.setGlobalActionHandler(ActionFactory.DELETE.getId(),
        registry.getAction(ActionFactory.DELETE.getId()));
    bars.updateActionBars();

    getViewer().setContextMenu(contextMenuProviderFactory.
        createContextMenuProvider(getViewer(), registry));

    KeyHandler keyHandler = new KeyHandler();
    keyHandler.put(KeyStroke.getPressed(SWT.DEL, SWT.DEL, 0),
        registry.getAction(ActionFactory.DELETE.getId()));
    getViewer().setKeyHandler(keyHandler);
  }

  public void setContextMenuProviderFactory(
      ContextMenuProviderFactory contextMenuProviderFactory) {
    this.contextMenuProviderFactory = contextMenuProviderFactory;
  }

  public void setEditPartFactory(EditPartFactory editPartFactory)
  {
    this.editPartFactory = editPartFactory;
  }

  public void setGraphicalEditor(GraphicalEditor editor) {
    this.editor = editor;
  }

  public void updateContents() {
    getViewer().setContents(
        (Object) getEditorResource(Object.class));
  }
}
```

Listing 64: Implementation a generic outline view using the graphical editor

Create and initialize

Like you already noticed in the listing above, we had to implement the configuration and initialization of the outline page. In the method `createControl` we had to create the viewable components. As an easy view we used the `SashForm`, which lays out its children either in a row or column arrangement and places a sash between them. This must be passed to the `createControl` method of the viewer.

In a further step we set the `EditDomain`, the `EditPartFactory` and the content, which should be displayed. We got all of them through the `getAdapter` method of the graphical editor. We decided for this solution, because it's a common way widely used in the Eclipse RCP. Otherwise we had to define an interface providing methods for those resources and to implement it in the graphical editor.

At last we synchronized the selection between the outline and the graphical editor. Additionally we also registered the global action handlers in the `init` method to allow also manipulations using the toolbar and menu entries. Unfortunately using the extension points for their definition did not work, so we had to implement it in this case.

Disposal

Finally in the `dispose` method we had to deregister the synchronization, before the outline will be disposed and removed. Otherwise the outline page would not be disposed properly and remains in the memory.

EditPartFactory

Like mentioned in the overview of this chapter, GEF uses the same concept for the elements as in the graphical editor (see chapter 6.3.8.4) also for the outline page. Therefore we had to implement a factory creating the edit parts for the given model objects. For this purpose the already known `EditPartFactory` can be used.

But instead of creating graphical edit parts, it creates in this case tree based edit parts. The rest of this class is similar to the existing already one.

EditParts

Also very similar is the implementation of the edit parts. We must inherit from `AbstractTreeEditPart` instead of the graphical variant.

We also had to register and deregister the `PropertyChangeListener` on the model object we want to be notified about changes, using the methods `activate` and `deactivate`. Like in the graphical variant, we located them in an abstract super class to reduce duplicate code, like shown in **Listing 65** below.

```
public abstract class AbstractNodeTreeEditPart extends Ab-
stractTreeEditPart {

  @Override
  public void activate() {
    super.activate();

    Node model = (Node) getModel();
    model.addPropertyChangeListener(this);
  }

  @Override
  public void deactivate() {
    Node model = (Node) getModel();
    model.removePropertyChangeListener(this);
```

```
    super.deactivate();
  }
}
```

Listing 65: Implementation of an abstract edit part containing common parts used in the outline tree view

Additionally we had to override the methods `getModelChildren` and `createEditPolicies`. The first method returns a list with the children, who will be shown as sub elements, and the second registers edit policies handling the user interactions. Their implementations are similar to the ones in the graphical variant.

The only different thing is the visual part. We don't have to create a figure like in the graphical edit parts, because the `TreeItem` will be created automatically. So we could simply use them for example in the `refreshVisuals` method to fill text and an appropriate icon.

The following code snippet in **Listing 66** shows the implementation of a `TreeEditPart` with the different methods.

```
public class PersonAttributeTreeEditPart extends AbstractNodeTreeEditPart {

  @Override
  protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE,
        new NodeEditPolicy());
  }

  public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getPropertyName().equals(Diagram.PROPERTY_ADD) ||
        evt.getPropertyName().equals(Diagram.PROPERTY_REMOVE)) {
      refreshChildren();
    } else if (evt.getPropertyName().equals(
                  Node.PROPERTY_NODE_NAME)) {
      refreshVisuals();
    }
  }

  @Override
  protected void refreshVisuals() {
    GPersonAttribute model = (GPersonAttribute) getModel();

    setWidgetText(model.getName());
    setWidgetImage(PlatformUI.getWorkbench().getSharedImages().
        getImage(ISharedImages.IMG_OBJ_ELEMENT));
  }
}
```

Listing 66: Example implementation of an edit part used in the outline tree view

Graphical editor

The last step was the integration in the graphical editor. We had to instantiate the outline page and return it in the `getAdapter` method using the interface `IContentOutlinePage`, like shown in the excerpt (**Listing 67**) below.

```
private ContentOutlinePage outlinePage =
    new DefaultOutlinePageWithMiniatureView(this);
```

```
...
...

@Override
@SuppressWarnings("unchecked")
public Object getAdapter(Class type) {
  if (type == IContentOutlinePage.class) {
    return outlinePage;
  } else if (type == EditDomain.class) {
    return getEditDomain();
  } else if (type == SelectionSynchronizer.class) {
    return getSelectionSynchronizer();
  } else if (type == ZoomManager.class) {
    return ((ScalableRootEditPart) getGraphicalViewer()
        .getRootEditPart()).getZoomManager();
  } else if (type == Object.class) {
    return contents;
  }

  return super.getAdapter(type);
}
```

Listing 67: Excerpt out of the graphical editor to integrate the outline view

Based on the decision above, to get the required resources for the outline page over the `getAdapter` method of the graphical editor, we had to extend it further. We additionally added the return statements for the `EditDomain`, the `SelectionSynchronizer` and the `Object`, representing the diagram content.

The `ActionRegistry` and the `GraphicalViewer` are already returned from a super class provided by GEF. Therefore to invoke the `getAdapter` method of the super class is enough.

6.3.8.10.3 Problems

Difference between model and visualization structure

We tried to use a more encapsulated structure for the visualization of the model elements than they are hold there. So we tried to manipulate the return values of the method `getModelChildren` in the different edit parts. Unfortunately we got some exceptions.

The problem was mainly a cycle. In an edit part we returned a filtered list, but we forgot to remove the model object which is represented through it. This caused a never ending cycle. After the correction of this mistake, the visualization worked like expected.

But the next problem occurred very quickly. Through the change of the shown structure, also the return values of the `getHost` method in the edit policies changed. So we had to implement an additional edit policy, because the already existing was still used for the edit part in the graphical editor.

And this change caused further problems with the update of the outline view, which doesn't happen properly. So we decided to undo these changes.

Update the outline data

An additional problem we had during the implementation of the graphical editor was the update of the content displayed in the outline page. It occurred after the integration of the graphical editor in the application, which manages the domain model in a central service. Changes on it caused notifications through which the

graphical editor is able to read its input once again. This worked like expected.

Different was the outline page. Although it is able to get the content it should display through the graphical editor's `getAdapter` method, it was not notified about the changes.

A possible solution for this problem was to register the outline page also, so it is able to react to the notifications. But this would lead to additional dependencies, which we tried to avoid, because its implementation is very generic. Especially in combination with Spring's dependency injection.

So we decided to implement an abstract class `ExtContentOutlinePage` inheriting from `ContentOutlinePage`. In this class we defined an abstract method `updateContents`, whose responsibility is to update the outline page's content, like shown in **Listing 68** below.

```
public abstract class ExtContentOutlinePage extends ContentOutli-
nePage {

  public ExtContentOutlinePage(EditPartViewer viewer) {
    super(viewer);
  }

  public abstract void updateContents();
}
```

Listing 68: Implementation of the abstract class `ExtContentOutlinePage`

Since we hold the instance of the outline page in the graphical editor, we could only add the call of this method after the update of the editor's content.

6.3.8.11 Miniature View

Overview

An additional feature for providing an overview about the diagram in the graphical editor is the miniature view. It provides a small image showing the whole diagram and the selection of visible part.

GEF provides for the miniature view not as a special component, like for other features, but it provides the classes to implement it easily.

6.3.8.11.1 Our Solution

Implementation

We implemented the miniature view on the one hand as an additional part in the outline view and on the other hand also as an independent view. We tried both ways, to provide the developer and the user the possibility to choose if they want a miniature view and where they want it.

As component in the outline view

For the extension of the outline view, we created a subclass of the already existing implementation, because we want to provide both classes, one with the miniature view and one without it.

In the new subclass `DefaultOutlinePageWithMiniatureView` we had to override the `createControl` method to create the miniature view and the

`dispose` method to remove and destroy it, like show in **Listing 69**.

```
public class DefaultOutlinePageWithMiniatureView
                                        extends DefaultOutlinePage {

  private ScrollableThumbnail thumbnail;

  private DisposeListener disposeListener = new DisposeListener()
  {
    public void widgetDisposed(DisposeEvent e) {
      if (thumbnail != null) {
        thumbnail.deactivate();
        thumbnail = null;
      }
    }
  };

  @Override
  public void createControl(Composite parent) {
    super.createControl(parent);

    // Miniature view
    Canvas canvas = new Canvas(sash, SWT.BORDER);
    LightweightSystem system = new LightweightSystem(canvas);

    GraphicalViewer viewer =
        getEditorResource(GraphicalViewer.class);
    ScalableRootEditPart rootEditPart =
        (ScalableRootEditPart) viewer.getRootEditPart();

    thumbnail = new ScrollableThumbnail(
        (Viewport) rootEditPart.getFigure());
    thumbnail.setSource(rootEditPart
        .getLayer(LayerConstants.PRINTABLE_LAYERS));

    system.setContents(thumbnail);

    viewer.getControl().addDisposeListener(disposeListener);
  }

  @Override
  public void dispose() {
    GraphicalViewer viewer =
        getEditorResource(GraphicalViewer.class);

    if (viewer.getControl() != null &&
        !viewer.getControl().isDisposed()) {
      viewer.getControl().removeDisposeListener(disposeListener);
    }

    super.dispose();
  }
}
```

Listing 69: Implementation of miniature view integrated in the outline

For the creation of the miniature view, we had to define a new `Canvas` and to encapsulate in a `LightweightSystem` to provide the container for its visualization. Further we got the `ScalableRootEditPart` from the graphical editor, to get the figure displaying the whole diagram. This can now be added to the `LightweightSystem` for a proper display.

Additionally we had to register a `DisposeListener`, to clean up the miniature view properly in the case when the view will be closed.

As an independent view

The implementation of the miniature as an independent view is very similar to the one integrated in the outline view. For its implementation we used the `ContentOutlinePage`, but omitted the statements for the conventional outline.

In this class we also had to add to pass the graphical editor, like in the conventional outline view, because we used some of the attributes it manages. But differently we don't have to set the `EditDomain` and the other factories.

6.3.8.11.2 Problems

Miniature as an own view

We tried to implement the miniature view as class inheriting either from `Page` or `ViewPart`, like described above. Additionally, we had to configure the extension points `org.eclipse.ui.views` to register the new view and `org.eclipse. ui.perspectiveExtensions` to add it by default to the perspective. In the last step we extended the `getAdapter` method in the graphical editor to return a new instance of this class.

Unfortunately we got exceptions, because Eclipse RCP was not able to create the new view, although we implemented all required parts. We could not solve this problem, because Eclipse RCP seems not able to handle outline-like views from the graphical editor.

6.3.8.12 Action Bar

Overview

Normally the graphical editor requires additional entries in the existing menu- and toolbars to provide the user with tools for the manipulation of the diagram with its elements. In the extension point `org.eclipse.ui.editors`, RCP allows the configuration of a contributor class, whose purpose is to configure and extend the menu- and toolbars.

For this purpose, GEF provides the abstract class `ActionBarContributor` to simplify its implementation through specific hook-in methods. It allows accessing global entries as well as creating and contributing custom menu- and toolbars.

6.3.8.12.1 Our Solution

Implementation

In our solutions we decided to implement the action bars in the `ActionBarContribution` class. The reason for this decision were mainly problems with the extension points (see chapter 0 for further details), which doesn't seem to work.

Accessing global action bar entries

To allow the use of actions in the graphical editor defined in the global menu- and toolbar, it is required to announce its usage. For this purpose the `ActionBarContributor` provides the hook method `declareGlobalActionKeys`. Through the method `addGlobalActionKey` passing the id of the actions we

intended to use, we made this announcement.

The following excerpt out of the `ActionBarContributor` class shows such an announcement for the actions *Save*, *Redo* and *Undo*.

```
@Override
protected void declareGlobalActionKeys() {

  addGlobalActionKey(ActionFactory.UNDO.getId());
  addGlobalActionKey(ActionFactory.REDO.getId());
  addGlobalActionKey(ActionFactory.SAVE.getId());
}
```

Listing 70: Declaration of the global action keys in the `ActionBarContributor` of the graphical editor

Contributing editor based action bars

More different is the implementation and configuration of the view specific action bar entries. We had to implement the methods `buildActions` and `contributeTo...` in our `ActinBarContributor` class. In the first method we had to create and initialize the actions which we intended to use, like **Listing 71** shows. Through these directives, they will be managed by our contributor.

```
@Override
protected void buildActions() {
  addRetargetAction(new DeleteRetargetAction());

  addRetargetAction(new UndoRetargetAction());
  addRetargetAction(new RedoRetargetAction());

  addRetargetAction(new ZoomInRetargetAction());
  addRetargetAction(new ZoomOutRetargetAction());
}
```

Listing 71: Creation of editor specific actions in the `ActionBarContributor`

In the further methods the contribution of these actions to the corresponding action bar was necessary. Otherwise the registered actions would not be visible. The excerpt in **Listing 72** below shows the contribution of them to the toolbar.

```
@Override
public void contributeToToolBar(IToolBarManager toolBarManager) {

  toolBarManager.add(getAction(ActionFactory.DELETE.getId()));
  toolBarManager.add(new Separator());
  toolBarManager.add(getAction(GEFActionConstants.ZOOM_IN));
  toolBarManager.add(getAction(GEFActionConstants.ZOOM_OUT));
  toolBarManager.add(new ZoomComboContributionItem(getPage()));
}
```

Listing 72: Contribution of the actions to the editor specific action bar in the `ActionBarContributor`

6.3.8.12.2 Problems

Global action bar entries not enabled

A very handsome problem was the missing activation of the action bar entries, although they are configured properly. Initially we could not find the cause of this problem, although our implementation was complete.

After a comprehensive internet research on this problem, we found some example implementations, where the actions were additionally registered in the `createActions` method of the graphical editor. After adding these statements, the activation worked very well. Unfortunately, in this solution the registration respectively the declaration of the actions out of the global action bars must be done in both, the graphical editor and its action bar contributor.

Through the refactoring of the global menu- and toolbars, we found the real cause of the problem. We created the global action bars through extension points; therefore the registration of the actions in the global action bar contributor was missing. But this is required in the way like shown in **Listing 73**. Otherwise this registration must be done somewhere else, for example in the graphical editor.

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor
{
  ...

  @Override
  protected void makeActions(IWorkbenchWindow window) {
    super.makeActions(window);

    register(ActionFactory.SAVE.create(window));
    register(ActionFactory.UNDO.create(window));
    register(ActionFactory.REDO.create(window));
    ...
  }
  ...
}
```

Listing 73: Registration of the actions in the global action bar

Extension Points

We also tried to configure the action bars depending on the graphical editor through extension points. Although this should be possible, we were not able to replace the implementation with them. So we gave up trying to replace them through an extension point declaration.

6.3.8.13 Context Menu

Overview

Additionally to the palette and the action bars, GEF allows also the usage of context menus in the graphical editor. Its development is very similar to those of the action bars.

For this purpose GEF provides the abstract class `ContextMenuProvider`, which allows a compact implementation.

6.3.8.13.1 Our Solution

ContextMenu-Provider

We implemented the context menus of our graphical editors like proposed with the `ContextMenuProvider`. The inheritance from this abstract class forced us to override the method `buildContextMenu`.

In this method we got the different actions from the `ActionRegistry`, we passed as a parameter in the constructor. Afterwards, we added them to the `IMenuManager` passed as parameter, like shown in **Listing 74** below.

```
public class DefaultContextMenuProvider extends ContextMenuPro-
vider {

  private ActionRegistry actionRegistry;

  public DefaultContextMenuProvider(EditPartViewer viewer,
        ActionRegistry registry) {

    super(viewer);
    this.actionRegistry = registry;
  }

  @Override
  public void buildContextMenu(IMenuManager menu) {
    IAction action;

    GEFActionConstants.addStandardActionGroups(menu);

    action = actionRegistry.getAction(
        GEFActionConstants.ZOOM_IN);
    menu.appendToGroup(GEFActionConstants.GROUP_VIEW, action);

    action = actionRegistry.getAction(
        GEFActionConstants.ZOOM_OUT);
    menu.appendToGroup(GEFActionConstants.GROUP_VIEW, action);

    action = actionRegistry.getAction(
        ActionFactory.UNDO.getId());
    menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);

    action = actionRegistry.getAction(
        ActionFactory.REDO.getId());
    menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);

    action = actionRegistry.getAction(
        ActionFactory.DELETE.getId());
    menu.appendToGroup(GEFActionConstants.GROUP_EDIT, action);
  }
}
```

Listing 74: Implementation a context menu for the graphical editor

We required the `ActionRegistry` instance, because we had to get the actions from it. Otherwise we had to create new actions or get them in a different way. But this was not an appropriate solution why we decided for the simple parameter passing.

Factory

Additionally to the class extending `ContextMenuProvider`, we decided to implement a factory for its creation. The reason for this decision is to allow in a later step its configuration and injection with Spring. Since this is not possible in

a direct way, because its instantiation requires resources out of the graphical editor, we had to implement a redirection, our factory.

For this purpose we defined an interface for a common handling of the concrete implementations. This is very simple, defining only the method `createContextMenuProvider` with the parameters required for the creation of the `ContextMenuProvider`. The **Listing 75** below shows our interface.

```
public interface ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
      EditPartViewer viewer, ActionRegistry registry);
}
```

Listing 75: Definition of the `ContextMenuProviderFactory`

Its implementation is as simple as the interface itself. In the typical case, we only had to create a new instance of the `ContextMenuProvider` class and returned it. The **Listing 76** below shows such an implementation.

```
public class DefaultContextMenuProviderFactory implements
    ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
      EditPartViewer viewer, ActionRegistry registry) {
    return new DefaultContextMenuProvider(viewer, registry);
  }
}
```

Listing 76: Implementation of a typical `ContextMenuProviderFactory`

Graphical Editor

The last step of the context menu development was its integration in the graphical editor and the outline page.

In the graphical editor we added the instance of the context menu to the `GraphicalViewer` in the method `configureGraphicalViewer`, like shown in **Listing 77** below.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  ...

  // Context Menu
  viewer.setContextMenu(contextMenuProviderFactory
      .createContextMenuProvider(viewer, getActionRegistry()));

  ...
}
```

Listing 77: Integration of the context menu in the graphical editor

Outline page

Very similar was also the integration in the outline page. We added the instance to the viewer, accessible through the method `getViewer`, in the method `init`, instead of `configureGraphicalViewer`. The rest was equals.

6.3.8.13.2 Problems

Using extension points

As for the other action bars and context menus in our applications, we tried to configure and implement the context menu of the graphical editor through the extension point `org.eclipse.ui.menus`.

Although we configured it like the context menu for the other components, it was not displayed in the graphical editor. We also tried to add them to the application wide context menu using the Location URI `popup:org.eclipse.ui.popup.any`. Even in this case, the context menu was not displayed in the graphical editor, only in all other components.

So we did the configuration without the extension points and concentrated on a clean implementation.

6.3.8.14 KeyHandler

Overview

Beside the conventional methods to edit the elements in the diagram using the palette, the menus and toolbars, also interactions using the keys are very useful. This allows for example to delete the selected element using just the *Delete* key.

GEF allows such an interaction with the graphical editor and provides for this purpose the class `KeyHandler`. It can be used for the registration of `IActions` out of the `ActionRegistry`, which will be invoked for a defined key type.

6.3.8.14.1 Our Solution

Implementation

We implemented the registration of the key handlers for the graphical editor in a strategy outside the editor itself. The graphical editor invokes the concrete strategy implementation and passes itself to allow a proper registration.

The reason for this decision is mainly to hold the implementation of the graphical editor as compact as possible and to still allow a flexible registration of key handlers, which can be exchanged easily. For this purpose we defined a common interface `KeyHandlerFactory`, which contains two methods for the creation of a `KeyHandler`, one for the graphical editor and one for the outline page. **Listing 78** below shows the code snippet with the interface.

```
public interface KeyHandlerFactory {

  public KeyHandler createKeyHandler(GraphicalEditor editor);

  public KeyHandler createOutlineKeyHandler(
      ActionRegistry registry);
}
```

Listing 78: Definition of the `KeyHandlerFactory` for the creation of KeyHandlers

Registration of KeyHandlers

In the class implementing this interface, we had to create in a first step a new instance of `KeyHandler`. We used the `GraphicalViewerKeyHandler`, because it already provides default keystrokes for the common navigation in the graphical editor. So we could concentrate on the specific keystrokes for our editor.

Additionally to the already existing ones, we registered the keystroke for the delete key as well as for the '+' and '-' allowing to zoom in and out. The following **Listing 79** shows such an implementation.

```
public class DefaultKeyHandlerFactory implements KeyHandlerFacto-
ry {

  public KeyHandler createKeyHandler(GraphicalEditor editor) {

    GraphicalViewer viewer = (GraphicalViewer)
        editor.getAdapter(GraphicalViewer.class);
    ActionRegistry registry = (ActionRegistry)
        editor.getAdapter(ActionRegistry.class);

    KeyHandler keyHandler =
        new GraphicalViewerKeyHandler(viewer);

    keyHandler.put(KeyStroke.getPressed(SWT.DEL, SWT.DEL, 0),
        registry.getAction(ActionFactory.DELETE.getId()));
    keyHandler.put(KeyStroke.getPressed('+', SWT.KEYPAD_ADD, 0),
        registry.getAction(GEFActionConstants.ZOOM_IN));
    keyHandler.put(KeyStroke.getPressed('-', SWT.KEYPAD_SUBTRACT,
        0), registry.getAction(GEFActionConstants.ZOOM_OUT));

    return keyHandler;
  }

  public KeyHandler createOutlineKeyHandler(
      ActionRegistry registry) {

    KeyHandler keyHandler = new KeyHandler();

    keyHandler.put(KeyStroke.getPressed(SWT.DEL, SWT.DEL, 0),
        registry.getAction(ActionFactory.DELETE.getId()));

    return keyHandler;
  }
}
```

Listing 79: Implementation of the `KeyHandlerFactory` for the creation of KeyHandlers

Graphical Editor

After the implementation of the `KeyHandlerFactory` to create the `KeyHandler`s, we had to integrate it in our graphical editor. The method `configureGraphicalViewer` is the proposed location for such configurations, so we integrated it in this method. The integration itself was very easy, like the **Listing 80** shows.

```
@Override
protected void configureGraphicalViewer() {
```

```
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();
  ...

  // KeyHandler definition
  viewer.setKeyHandler(keyHandlerFactory.createKeyHandler(this));
}
```

Listing 80: Integration of the KeyHandlers in the graphical editor

Outline page

Very similar was also the integration in the outline page. The only difference is the method where we implemented it. Instead the `configureGraphical-Viewer` method, which does not exist, we implemented the integration in the `init` method.

6.3.8.15 ZoomManager

Overview

An important feature of today's applications is the zoom to allow the visualization in different sizes, depending on the user's choice. This applies especially for graphical visualizations like the graphical editor.

For this purpose GEF allows an easy integration of such a feature with less effort. Therefore it provides abstract classes and default implementations, like the `ScalableRootEditPart` or the `ZoomManager`.

6.3.8.15.1 Our Solution

Implementation

In our applications we also implemented the zoom feature, because in today's applications it's a must requirement.

Like other components too, we implemented the creation and configuration of the `ZoomManager` in a strategy outside the graphical editor itself. The graphical editor invokes the concrete strategy implementation and passes itself to allow a proper registration.

The reason for this decision is also mainly the same. We intended to hold the implementation of the graphical editor as compact as possible. This allowed us to implement a configuration of the `ZoomManager` in a very flexible manner which can be exchanged easily. For this purpose we defined a common interface `ZoomManagerFactory`, which contains only a single method. **Listing 81** below shows a code snippet with our interface.

```
public interface ZoomManagerFactory {

  public ScalableRootEditPart createZoomManager(GraphicalEditor
editor);
}
```

Listing 81: Definition of the `ZoomManagerFactory` for the configuration of the ZoomManager

ZoomManager

Afterwards we had to develop the class implementing the `ZoomManagerFactory`. In a first step we had to create a new instance of `ScalableRootEditPart` as the new visual root object. This is necessary because it's the class which uses the `ZoomManager` and therefore is responsible to provide it.

On the instance of the `ZoomManager` we could do now the required configurations, like setting the zoom levels or the zoom level contributions as well as linking it with the editor and its actions. The **Listing 82** shows such an implementation.

```
public class DefaultZoomManagerFactory implements ZoomManagerFactory {

  public ScalableRootEditPart createZoomManager(
        GraphicalEditor editor) {

    ScalableRootEditPart rootEditPart =
        new ScalableRootEditPart();
    ZoomManager manager = rootEditPart.getZoomManager();

    manager.setZoomLevels(new double[] {
        0.25, 0.5, 0.75, 1.0, 1.5, 2.0,
        2.5, 3.0, 4.0, 5.0, 10.0, 20.0 });

    ArrayList<String> zoomContributions =
        new ArrayList<String>();
    zoomContributions.add(ZoomManager.FIT_ALL);
    zoomContributions.add(ZoomManager.FIT_HEIGHT);
    zoomContributions.add(ZoomManager.FIT_WIDTH);
    manager.setZoomLevelContributions(zoomContributions);

    registerZoomManagerActions(editor, manager);
    registerZoomMouseHandler(editor);

    return rootEditPart;
  }

  private void registerZoomManagerActions(GraphicalEditor editor,
        ZoomManager manager) {
    ActionRegistry registry = (ActionRegistry) editor
        .getAdapter(ActionRegistry.class);

    registry.registerAction(new ZoomInAction(manager));
    registry.registerAction(new ZoomOutAction(manager));
  }

  private void registerZoomMouseHandler(GraphicalEditor editor) {
    GraphicalViewer viewer = (GraphicalViewer) editor
        .getAdapter(GraphicalViewer.class);

    viewer.setProperty(MouseWheelHandler.KeyGenerator.
        getKey(SWT.SHIFT), MouseWheelZoomHandler.SINGLETON);
  }
}
```

Listing 82: Implementation of the `ZoomManagerFactory` for the creation of a ZoomManager

Additionally we configured the `GraphicalViewer` to allow zooming with the mouse wheel in combination with the *Shift* key.

Graphical Editor

After the implementation of the `ZoomManagerFactory` to create and configure

the `ZoomManager`, we had to integrate it in our graphical editor. The method `configureGraphicalViewer` is the proposed location for such configuration, so we integrated it in this method. The integration itself was as easy as for the key handlers. **Listing 83** shows an excerpt of the graphical editor where the `ZoomManager` will be integrated.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();
  ...

  // ZoomManager definition
  viewer.setRootEditPart(
      zoomManagerFactory.createZoomManager(this));

  ...
}
```

Listing 83: Integration of the ZoomManager in the graphical editor

Action bars extension

Additionally to the integration of the zoom feature in the graphical editor, we also had to extend the action bars. Otherwise no controls would be displayed for its use.

Therefore we only had to add additional actions in the action bar like already shown in **Listing 71** and **Listing 72**. Its integration was very simple.

Very similar was also the extension of the context menu. Also in this case we had to create new actions and to add it afterwards to the corresponding menu. **Listing 74** already shows the integration of the *Zoom In* and *Zoom Out* controls in the context menu.

6.3.9 Text Editor

Overview

The main intention of this chapter is to describe our solution concerning the use of a text editor and the integration of the platform text facility provided by Eclipse.

Diagram

The diagram below describes the buildup of the text editor implementation in our solution. Please note that workbench configurations, like e.g. extensions through extension points, are not visible in the scope shown in the diagram.

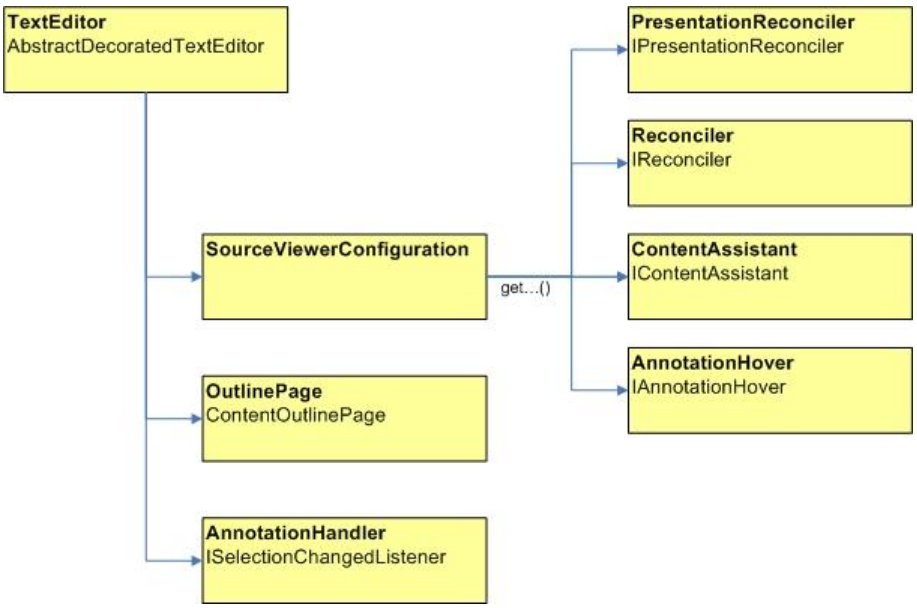

Figure 18: Buildup of the text editor implementation

Document Model

The Eclipse text facility provides a model called `IDocument` which describes the document model for text and is used to access the text in a text editor. It includes the following operations:

- Text manipulation
- Text positioning
- Text partitioning
- Line information
- Document change listeners
- Document partition change listeners

Document Providers

Eclipse provides default implementations of `IDocumentProvider`, which can be used to load content and parse an `IDocument` instance based on it. Because of the use of text file buffers (`ITextFileBuffer`), the `TextFileDocumentProvider` is the most appropriate one in order to load text based files.

The inclusion of a document provider is done by adding a document provider on the extension point `org.eclipse.ui.editors.documentProviders`. The code snippet below shows how a document provider is configured in our solution.

```
<extension
  point="org.eclipse.ui.editors.documentProviders">

  <provider
    class="org.eclipse.ui.editors.text.TextFileDocumentProvider"
    id="org.eclipse.ui.editors.text.StorageDocumentProvider"
    inputTypes="org.eclipse.ui.IStorageEditorInput">
  </provider>
</extension>
```

Listing 84: Code snippet showing the configuration of the document provider

Text Editor

The text editor itself is the widget provided by Eclipse supporting different configuration possibilities.

The `ITextEditor` interface describes the functional and configurationally capabilities of a text editor. Text editors use document providers to obtain access to the document model for a specific content.

To implement a text editor it is recommended to either subclass the `AbstractTextEditor` or `AbstractDecoratedTextEditor`. The decorated text editor adds the following functionalities to the basic implementation:

- Line numbers
- Change ruler
- Overview ruler
- Print margins
- Current line highlighting

Because we want to use these additional features, we extend the `AbstractDecoratedTextEditor` class for the implementation of our text editor.

Action Bar Contributor

The corresponding action bar contributor (`IEditorActionBarContributor`) defines the actions for one or more editors and adds them to a given action bars. Therefore it avoids the creation of duplicate actions and action images in the case multiple editors are used. The responsibility of the action bar contributor is to create the actions whereas the editor component is responsible to do their implementation. According to this structure, the action bar contributor can be shared by the open editors and there is one set of actions for one or more open editors.

In our solution we extend `BasicTextEditorActionContributor` and create an own implementation. This action bar contributor manages the installation and the removal of global actions for text editors of the same type.

It is configured in the *plugin.xml* descriptor, where a view defines its relationship to a contributor class through the definition of an extension on the `org.eclipse.ui.editors` extension point.

The code snippet in **Listing 85** shows how the contributor class is configured in an example application.

```
<extension point="org.eclipse.ui.editors">
  <editor
    class="org.herasaf.tutorial.personlocator.ui.
           texteditor.PersonLocatorTextEditor"
    contributorClass="org.herasaf.tutorial.personlocator.ui.
                      texteditor.PersonLocatorTextEditorContributor"
    default="true"
    extensions="plc"
    icon="icons/editor.jpg"
    id="org.herasaf.tutorial.personlocator.ui.texteditor"
    name="PersonLocatorTextEditor">
  </editor>
</extension>
```

Listing 85: Snippet of the configuration of the action contributor relationship with the view

Plug-in Depen-

The following list describes the required plug-ins for the implementation of the

dencies

text editor.

- org.eclipse.core.runtime
- org.eclipse.jface.text
- org.eclipse.ui
- org.eclipse.ui.views
- org.eclipse.ui.editors
- org.eclipse.ui.workbench.texteditor

6.3.9.1 Source Viewer Configuration

Description

The `SourceViewerConfiguration` is a class bundling the configuration of a source viewer defined by the interface `ISourceViewer`. As seen in **Figure 18** the `SourceViewerConfiguration` provides different possibilities to configure the source viewer, which will be described in the following chapters.

Implementation

In our solution we use an own source viewer configuration, which is a subclass of `SourceViewerConfiguration`. It is created and set during the construction phase of the text editor.

The following code snippet illustrates a possible setup of the source viewer configuration in an example application.

```
public PersonLocatorTextEditor() {
  setSourceViewerConfiguration(
      new PersonLocatorSourceViewerConfiguration(
          this, getSharedColors()));
}
```

Listing 86: Snippet of the setup of the source viewer configuration in the text editor

As **Listing 86** shows, the source viewer configuration gets the text editor and the shared colors in the constructor as parameters. Details in the implementation of the source viewer configuration will be described in the following chapters.

Viewers

The `ITextViewer` interface describes the connection between a text widget and an `IDocument`, whereas the `IDocument` is used as the widgets text model. The text viewer supports basic text processing functionality, like e.g. view port listeners, which inform about changes in the visible area, text listeners, which inform about changes in the document, the configuration of an undo manager and further more.

`ISourceViewer` extends the `ITextViewer` interface and adds additional source processing functionalities like visual annotations, visual range indications, syntax highlighting and content assistance. A source viewer is configured by calling the `configure` method and by passing the `SourceViewerConfiguration`.

In order to provide these features, we used the source viewer and defined, as mentioned above, an own source viewer configuration for its setup.

Detailed Description

Detailed information about the functionality and configuration support can be found in the Javadoc pages of `ITextViewer` and `ISourceViewer`.

6.3.9.2 Syntax highlighting

Description

Syntax highlighting is used in structured documents to make the text more readable by highlighting parts of the text independently from their position in the document.

Diagram

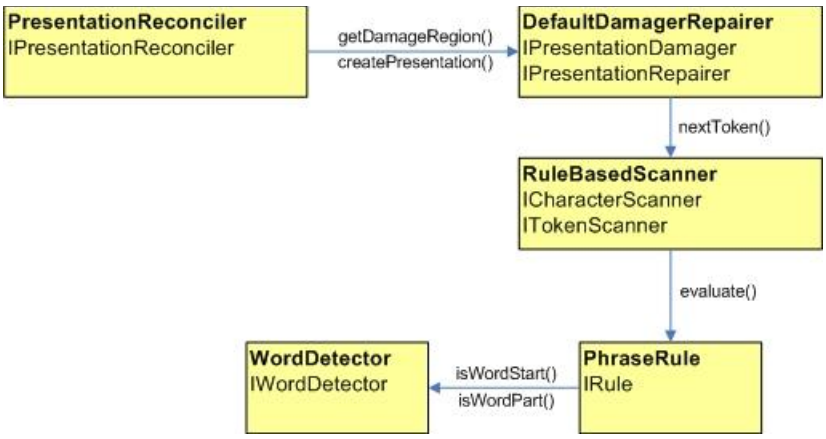

Figure 19: Structure of the syntax highlighting implementation in our solution

Implementation

In our own implementation of the `SourceViewerConfiguration` class we overrode the method `getPresentationReconciler`, which returns an `IPresentationReconciler`. This `IPresentationReconciler` consists of a damager (`IPresentationDamager`) and a repairer (`IPresentationRepairer`). The damager recognizes where the document has been damaged, whereas the repairer is responsible for the update and for the coloring of the damaged parts in the text.

The code snippet in **Listing 87** shows the setup of the damager and repairer in an example application.

```
public IPresentationReconciler getPresentationReconciler(
        ISourceViewer sourceViewer) {

  PresentationReconciler reconciler =
      new PresentationReconciler();
  DefaultDamagerRepairer dr =
      new DefaultDamagerRepairer(getScanner());

  reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
  reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

  return reconciler;
}
```

Listing 87: Code snippet showing the setup of the damager and the repairer

The framework provides a `DefaultDamagerRepairer`, which implements both, the damager interface as well as the repairer interface. The provided functionality of this default implementation was sufficient and there was no need to create an individual implementation in this scope. For the same reason we also use the `RuleBasedScanner` in our solution, which is also available in the framework.

Taking the same application as an example again, the next code snippet shows how the scanner and the rules are created.

```
private ITokenScanner getScanner() {
  RuleBasedScanner scanner = new RuleBasedScanner();

  IRule[] rules = new IRule[2];
  rules[0] = createPersonsWordRule();
  rules[1] = createLocationWordRule();

  scanner.setRules(rules);
  return scanner;
}
```

Listing 88: Code snippet showing the implementation of the scanner and the rules

For the definition of rules for the `RuleBasedScanner` there also are default implementations of rules, like e.g. `SingleLineRule` or `MultiLineRule`, available. In order to be able to recognize whole names and phrases, which also can include whitespaces, we created an own rule called `PhraseRule` since this functionality was not provided by a default implementation in the framework at the time this thesis was written. This is shown in the code snippet below.

```
private IRule createLocationWordRule() {
  IToken titleToken =
      new Token(new TextAttribute(
         fColors.getColor(new RGB(100, 100, 200)),
         null, SWT.BOLD)
     );

  PhraseRule phraseRule =
      new PhraseRule(new SimpleWordDetector(),
          titleToken, new ModelDataProvider(),
          ModelDataProvider.LOCATIONS);

  return phraseRule;
}
```

Listing 89: Code snippet showing the creation of the phrase rules

The scanner splits the text in tokens based on the configured rules and describes them with `TextAttributes`, which define the color and the overall appearance of the token.

6.3.9.3 Model reconciliation

Description

In structured documents, such as Java source files, there is a whole model standing behind the text. This model is derived by parsing the content out of the source, like e.g. a text file, and out of the active editor. In the lifecycle of a text

editor though, this model changes permanently due to the changes the user makes on the document during the typing.

Depending on the size of the document and the parsing algorithm, this action can take long time for the execution. Therefore the framework provides a reconciling infrastructure, which gives the possibility to run such long running tasks in own background threads.

Diagram

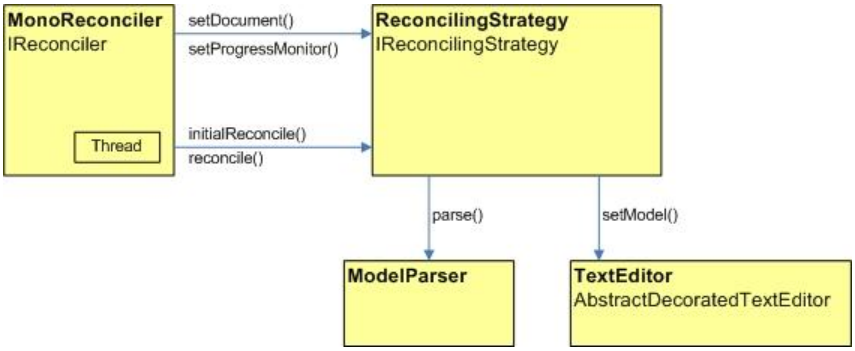

Figure 20: Diagram of the reconciling infrastructure in the text editors configuration

Implementation

By overriding the method `getReconciler` in the `SourceViewerConfiguration` an own reconciler implementing the interface `IReconciler` can be provided.

The code snippet in **Listing 90** shows the definition of a reconciling strategy in an example application.

```
public IReconciler getReconciler(ISourceViewer sourceViewer) {
  PersonLocatorReconcilingStrategy strategy =
      new PersonLocatorReconcilingStrategy(fEditor);

  MonoReconciler reconciler =
      new MonoReconciler(strategy, false);
  reconciler.setProgressMonitor(new NullProgressMonitor());
  reconciler.setDelay(500);

  return reconciler;
}
```

Listing 90: Code snippet showing the definition of a reconciling strategy

The reconciler is responsible for the management of the background thread and for the invocation of the reconciling strategy (`IReconcilingStrategy`). In our solution we use the `MonoReconciler`, which provides only one reconciling strategy. Because we did not split the document in different content types (partitions) and the document size is not very large, the `MonoReconciler` was more appropriate than the multiple strategies supporting `Reconciler` in this case. In order to give the reconciling strategy the possibility to show its progress and to be able to stop the process, a progress monitor (`IProgressMonitor`) is passed to the reconciler. Additionally, a delay is setup to keep the parsing process from running after every key press.

The reconciler passes the document through the `setDocument` method and the

progress monitor through the `setProgressMonitor` method to the reconciling strategy. Afterwards the methods `initialReconcile` and `reconcile` can be invoked. The reconciling strategy starts the parsing process in its `reconcile` methods and has access on the document and on the progress monitor to be able to complete its job. After the document has been parsed, the derived model is set on the text editor.

6.3.9.4 Outline Page

Description

The Eclipse platform provides an outline viewer, which shows an overview of the current active editor. The outline view will help the user not to lose the overview in a larger document.

Implementation

As soon as an editor is opened, the framework invokes the `getAdapter` method in the text editor and requests an `IContentOutlinePage`.

The outline view itself is implemented in an own class by extending the class `ContentOutlineView` and instantiated by the text editor which returns it.

The code snippet below shows how the outline page support is implemented in an example application.

```
public Object getAdapter(Class required) {
  if (IContentOutlinePage.class.equals(required)) {
    if (fOutlinePage == null)
      fOutlinePage = new PersonLocatorOutlinePage(this);
    return fOutlinePage;
  }
  if (fProjectionSupport != null) {
    Object adapter =
        fProjectionSupport.getAdapter(getSourceViewer(),
            required);
    if (adapter != null)
      return adapter;
  }
  return super.getAdapter(required);
}
```

Listing 91: Code snippet showing the implementation of the outline view support

The abstract class `ContentOutlineView` provides a `TreeViewer` which can be used to show the outline by defining an own implementation of a `ContentProvider` and a `LabelProvider`.

The `TreeViewer` receives the model as an input through the method `setInput`. Having an individual object representing the model of the application, the `TreeViewer` has to be configured on how to handle and show the data. An implementation of a `ContentProvider` defines how the model content is returned in a tree-structured view, whereas an implementation of `LabelProvider` returns the text and the icons to be shown in the tree.

6.3.9.5 Content Assistance

Description

Content assistance is used to show contextual completion proposals in form of a list in the text editor. This helps the user to reduce typing effort and avoid typing errors.

Implementation

In the `SourceViewerConfiguration` the content assistance can be configured by overriding the method `getContentAssistant` and returning a class implementing `IContentAssistant`.

The framework provides a class `ContentAssistant`, which is a default implementation of `IContentAssistant`. The `ContentAssistant` requires an `IContentAssistProcessor` to compute the completion proposals.

The code snippet in **Listing 92** shows how the content assistance is implemented in an example application.

```
public IContentAssistant getContentAssistant(ISourceViewer sour-
ceViewer) {
  ContentAssistant assistant = new ContentAssistant();

  IContentAssistProcessor processor =
      new PersonLocatorCompletionProcessor(fEditor);
  assistant.setContentAssistProcessor(processor,
      IDocument.DEFAULT_CONTENT_TYPE);

  assistant.setContextInformationPopupOrientation(
      IContentAssistant.CONTEXT_INFO_ABOVE
  );

  assistant.setInformationControlCreator(
      getInformationControlCreator(sourceViewer)
  );
  assistant.enableAutoInsert(true);

  return assistant;
}
```

Listing 92: Code snippet showing the implementation content assistance

In our solution, we use an own implementation of `IContentAssistProcessor`, which overrides the `computeContextInformation` method and returns appropriate content to be shown as completion proposals in the text editor.

To be able to call the completion proposals by key press (CTRL+Space) a `TextOperationAction` has to be set in the implementation of the text editor.

The code snippet below shows how the actions are created in an example application.

```
protected void createActions() {
  super.createActions();

  IAction action =
    new TextOperationAction(
    PersonLocatorEditorPlugin.getDefault().getResourceBundle(),
          "ContentAssistProposal.",
          this, ISourceViewer.CONTENTASSIST_PROPOSALS
    );
```

```
  action.setActionDefinitionId(
      ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS
  );

  setAction("ContentAssist.", action);
  markAsStateDependentAction("ContentAssist.", true);
}
```

Listing 93: Code snippet showing the creation of the actions in the text editor

At the construction of the contributor class, a `RetargetTextEditorAction` is created, which is used to setup placeholders for actions in the menus or action bars and can be retargeted to dynamically changing actions from the active editor. In the initialization phase of the contributor the `RetargetTextEditorAction` is set as the global action handler in the action bars (`IActionBars`). The code snippet below shows how the implementation in an example application is done.

```
public class PersonLocatorEditorContributor
         extends BasicTextEditorActionContributor {

  ...

  private RetargetTextEditorAction fContentAssist;

  public PersonLocatorEditorContributor() {
    super();
    fContentAssist =
        new RetargetTextEditorAction(
            PersonLocatorEditorPlugin.
                getDefault().getResourceBundle(),
            "ContentAssistProposal."
        );
    fContentAssist.setActionDefinitionId(
        ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS);
  }

  public void setActiveEditor(IEditorPart part) {
    super.setActiveEditor(part);
    ITextEditor editor =
       (part instanceof ITextEditor) ? (ITextEditor) part : null;

    fContentAssist.setAction(
        getAction(editor, CONTENTASSIST_ACTION)
    );
  }

  public void init(IActionBars bars, IWorkbenchPage page) {
    super.init(bars, page);
    bars.setGlobalActionHandler(
        CONTENTASSIST_ACTION, fContentAssist
    );
  }

  ...
}
```

Listing 94: Code snippet showing the implementation of the contributor class

The resources for the content assistance are contained in the resource bundle,

which is a property file containing the provided messages. The content of such a file in an application is shown in the code snippet below.

```
ContentAssistProposal.label=Content Assist
ContentAssistProposal.tooltip=Content Assist
ContentAssistProposal.description=Content Assist
```

Listing 95: Code snippet showing the content of the resource bundle

6.3.9.6 Annotations

Description

The model behind the text editor gives further possibilities, like the recognition of errors in the source text, the addition of extra information and the execution of actions on the text. This is done with annotations.

Diagram

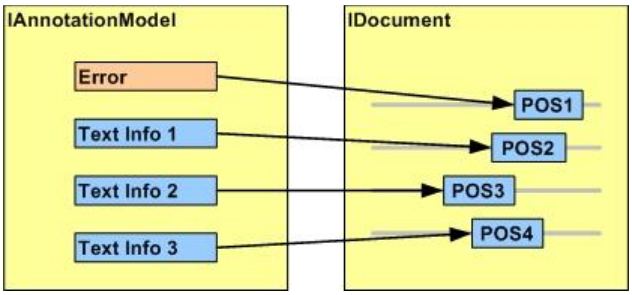

Figure 21: Diagram describing how annotations connect with the document model

Annotation Model

Beside the `IDocument` model, which is used to access the text, the annotation model (`IAnnotationModel`) holds extra information about the text. Thus an annotation is an extra part of information for text in a specific position (see **Figure 21**).

A `Position` describes through an offset and a length the position of the annotation in the text. The registered positions in the `IDocument` will be updated on document changes so that they always refer to that part of text they were intended to.

The text editor shows the annotations, depending on their type, very differently. They can be a mark with an icon on the left ruler of the editor, a mark in the right overview ruler or directly in the text as a squiggly mark and a background coloring.

Implementation

For the implementation of annotations we use an annotation handler, which is an implementation of `ISelectionChangeListener`. This annotation handler is registered on the `IPostSelectionProvider` of the text editor and updates on every `selectionChanged` event the annotation model. In our solution we use the `IPostSelectionProvider` because it sends the selection events with a delay and therefore avoids a flashing editor when e.g. the user holds an arrow key down to browse through the document.

The code snippet below shows how the annotation handler is registered on the editor in a sample application.

```
public PersonLocatorAnnotationHandler(PersonLocatorEditor editor)
{
    ((IPostSelectionProvider) editor.getSelectionProvider())
        .addPostSelectionChangedListener(this);

    fEditor = editor;
}
```

Listing 96: Snippet showing the registration of the annotation handler in the constructor

The annotation handler is created in the `createPartControl` method of the text editor and gets manually updated every time the model in the text editor is set. The code snippets below show how the annotation handler is created and updated manually on model changes.

```
public void createPartControl(Composite parent) {
    super.createPartControl(parent);

    ...

    fAnnotationHandler =
        new PersonLocatorAnnotationHandler(this);
}
```

Listing 97: Snippet showing the creation of the annotation handler

```
public void setModel(Model model) {
    fModel = model;
    if (fOutlinePage != null)
        fOutlinePage.setModel(model);

    if (fAnnotationHandler != null)
        fAnnotationHandler.update(getSourceViewer());
}
```

Listing 98: Snippet showing the manual update of the annotation handler on model changes

Annotation Illustration

For the illustration of the annotations, the text editor reads the extensions of the extension point `org.eclipse.ui.editors.markerAnnotationSpecification`. Thus, through the definition of an extension in the *plugin.xml* descriptor one can specify how a specific type of annotations should be presented.

The code snippet below shows how the configuration for the highlighting of persons in the person locator example application is done.

```
<extension
    point="org.eclipse.ui.editors.markerAnnotationSpecification">
  <specification
    annotationType="personlocator.highlightannotation"
    verticalRulerPreferenceKey="highlight.rulers.vertical"
    textPreferenceKey="highlight.text"
    colorPreferenceKey="highlight.color"
    highlightPreferenceKey="highlight.background"
    textPreferenceValue="false"
    textStylePreferenceValue="UNDERLINE"
    overviewRulerPreferenceKey="highlight.rulers.overview"
    presentationLayer="4"
    highlightPreferenceValue="true"
    label="Person"
    icon="icons/person.gif"
```

```
    colorPreferenceValue="253,255,157"
    verticalRulerPreferenceValue="true"
    overviewRulerPreferenceValue="true"
    textStylePreferenceKey="highlight.text.style">
  </specification>
</extension>
```

Listing 99: Snippet showing the annotation marker specification in *plugin.xml*

6.3.9.7 Context Menu

Description

The `AbstractDecoratedTextEditor`, as used in our solution, provides by default an implementation of a context menu. This context menu contains different actions, like Undo/Redo, Cut/Copy/Paste, etc. However, adding custom actions in the context menu to make the use of the application more practical and useful can be reasonable in many cases.

This chapter describes our solution concerning the use of the context menu and the addition of custom context menu entries.

Defining commands

The commands are defined through the extension point `org.eclipse.ui.commands`. A command is simply the abstract idea of some code being executed. The effective handling of the code is done by handlers.

The extension point provides the definition of a category, which enables to divide the commands in different categories based on their concerns. We use this feature in our solution and specify for each main concern an own category.

The following code snippet shows how the commands and the command category are defined in an example application.

```
<extension
  point="org.eclipse.ui.commands">

  <category
    id="org.herasaf.tutorial.personlocator.ui
        .texteditor.commands"
    name="Texteditor General Commands">
  </category>

  <command
    categoryId="org.herasaf.tutorial.personlocator.ui
                .texteditor.commands"
    id="org.herasaf.tutorial.personlocator.ui
        .texteditor.commands.person"
    name="Person Command">
  </command>
  ...
</extension>
```

Listing 100: Snippet showing the definition of commands in *plugin.xml*

As already mentioned above, the concrete code of a command is contained in the command handler. Handlers are defined through the extension point `org.eclipse.ui.handlers`. A handler implements the `IHandler` interface or extends the `AbstractHandler` class. Commands can have multiple handlers, whereas handlers can be configured when they should be active and/or enabled through the `activeWhen` and `enabledWhen` clauses.

In our solution we used exclusively command handlers to handle context menu behavior in the text editor, although the use of actions is also possible. A detailed description about commands and editor actions is given beneath in chapter 6.3.5.

Adding context menu entries

The extension point `org.eclipse.ui.menus` allows the developer to make additions in the main menu, toolbar, context menu, etc. in form of commands.

In order to add new entries in the context menu a new `menuContribution` is added to this extension point. The commands, then, are added to the `menuContribution`, which guides the framework to create menu entries based on the given preferences. The following code snippet shows how this issue is solved in an example application.

```
<extension
  point="org.eclipse.ui.menus">

  <menuContribution
    locationURI="popup:org.eclipse.ui.popup.any">

    <command
      commandId="org.herasaf.tutorial.personlocator.ui
                 .texteditor.commands.person"
      icon="icons/add_person.jpg"
      label="Add Person"
      style="push">
    </command>
    ...
  </menuContribution>
</extension>
```

Listing 101: Snippet showing the declaration of the menu in *plugin.xml*

In our solution we specify *"popup:org.eclipse.ui.popup.any"* for the `locationURI` in the `menuContribution`. This guides the framework to add the specified commands to all registered context menus in the application. For a further description of the positioning of menu entries, we would like to refer to the chapter 8.3.9.6 in the developers guide in this document.

Editor actions for commands

The editor actions are declared through the extension point `org.eclipse.ui.editorActions`. We use this extension point based on the decision in chapter 6.3.5.1.1 to handle actions inside of the text editor.

The definition of an `editorContribution` specifies the concerned text editor on which the configurations take effect. An `action` definition contained in the `editorContribution` then refers to a class which implements `IEditorActionDelegate` and handles the action when it takes place. Additionally, the actions are linked to a command through the definition in `definitionId`. This allows an action to handle a command.

The following code snippet describes the solution in an example application.

```
<extension
  point="org.eclipse.ui.editorActions">

  <editorContribution
    id="org.herasaf.tutorial.personlocator.ui
```

```
        .texteditor.editorContribution"
    targetID="org.herasaf.tutorial.personlocator.ui.texteditor">

    <action
      class="org.eclipse.springframework
             .util.SpringExtensionFactory:personEditorAction"
      definitionId="org.herasaf.tutorial.personlocator.ui
                    .texteditor.commands.person"
      id="org.herasaf.tutorial.personlocator.ui
          .texteditor.action.person"
      label="Person Action"
      style="push">
    </action>
    ...
  </editorContribution>
</extension>
```

Listing 102: Snippet showing the declaration of the editor actions in *plugin.xml*

By linking the action to a command, the action acts as a command handler and gets called each time the appropriate command is invoked inside of the text editor. Outside of the text editor the framework will look-up for handler declarations and if there are none existing, the labels will be shown as disabled.

Editor actions are more concrete and in the context of a text editor than abstract command handlers, which can be used universally. In our solution, where the document and the text selection are leading parts for the handling of the commands, the use of actions would be appropriate, but we decided against it.

6.3.10 Undo/Redo

Overview

Beside the execution of operations, also their undo and redo at a later time is an important feature of today's applications. Through the "recording" of the operations, the user is able to undo them in the same order or even to redo them again.

Eclipse RCP provides the developers with interfaces and abstract classes to implement the undo/redo feature in straightforward and common way. The base of this feature is the Command Pattern *[GoF1995]* and its extension the Command Processor *[Buschmann1996]*, which is used for the implementation of the operation objects, also called commands, and the execution unit.

But the component is not homogenous like other ones. It consists of several parts with different responsibilities. One of them is the *JobManager* for the asynchronous execution of the operations.

Schematic diagram

The **Figure 22** below shows the dependencies of them schematically as well as their interactions. The trigger to start the execution of an operation as well as its undo and redo is the user, which occurs an action.

This leads to the creation of a command object (operation), which encapsulates the action execution. Through its execution encapsulated in a Job, it's possible to run it asynchronously. But this is an optional feature.

After the execution of the command objects (operations), they will be cached in a command stack. This allows their undo and redo in a later step, if the user wants to do that.

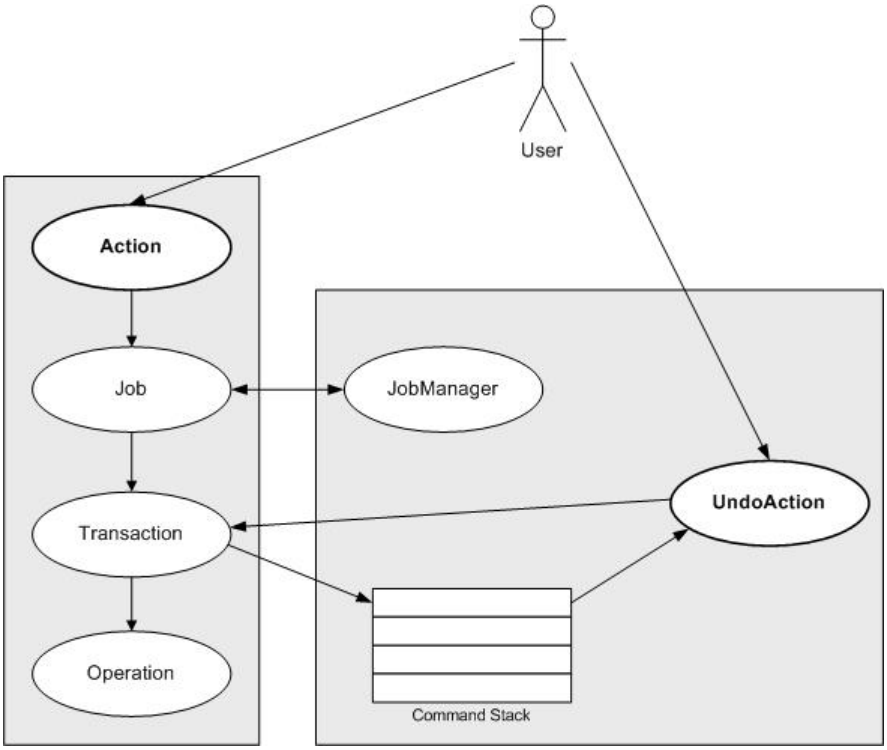

Figure 22: Schematic structure of the undo/redo component with the different parts

Like you have seen through the first impression of the undo/redo features, its general structure is given by RCP and can be varied only in details. So our solution described in the following subchapters follows this structure.

6.3.10.1 Our Solution

Structure

In our applications, we also used the undo/redo feature. For its implementation, apart from the one integrated in the graphical editor (see chapter 6.3.8), we used the given interfaces and abstract classes. The general structure is given by Eclipse's RCP, so we followed its recommendation and advices.

6.3.10.1.1 Operation

Operation

One of the key interfaces is `IUndoableOperation` which is used for the implementation of the command object. But we decided to use the `AbstractOperation` instead, because it provides default implementations for the defined methods. Typical methods we had to override to provide the required functionality were `execute` and `undo`.

We used additionally the `canExecute` and `canUndo` method to control the execution of the command object. This allowed us to separate the check, if the command object can be executed, from the execution itself. The advantages of such a solution are more focused methods and an easier maintenance.

The **Listing 103** below shows such a command object for the addition of two

integer values.

```
public class AddCalculationCommand extends AbstractOperation {

  private CalculatorModel model = null;
  ...

  public AddCalculationCommand(CalculatorModel model) {
    super("Add");
    this.model = model;
  }

  public void setOperand1(int operand1) {
    this.operand1 = operand1;
  }

  public void setOperand2(int operand2) {
    this.operand2 = operand2;
  }

  @Override
  public IStatus execute(IProgressMonitor monitor, IAdaptable
info)
    throws ExecutionException {

    int result = operand1 + operand2;
    model.setValue(result);
    return Status.OK_STATUS;
  }

  @Override
  public IStatus redo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException {

    return execute(monitor, info);
  }

  @Override
  public IStatus undo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException {

    model.setValue(oldValue);
    return Status.OK_STATUS;
  }
}
```

Listing 103: Example of a command object used to add two values.

Operation Execution

But this is only the command object itself. For its execution we implemented a method, which gets the command processor and passes the command object as parameter in its `execute` method. This is necessary for caching the executed command objects. Otherwise an undo and redo would not be possible.

To get the command processor, which is of type `IOperationHistory`, we had to get the `IWorkbench` from the `PlatformUI`. From this instance we could get the `IWorkbenchOperationSupport` and further the operation history, like shown in the code snippet in **Listing 104** below. On this object we could now execute the command object. But before, we decided to do some configurations, like setting the undo context or the maximum number of undo steps. This im-

proves the command processing.

```
public IStatus performOperation(IUndoableOperation op,
  IProgressMonitor monitor, IAdaptable info) {

  IWorkbench workbench = PlatformUI.getWorkbench();
  IWorkbenchOperationSupport operationSupport
                        = workbench.getOperationSupport();
  IOperationHistory operationHistory
                        = operationSupport.getOperationHistory();

  // Add the operation to the UndoContext
  op.addContext(operationSupport.getUndoContext());
  // Set the maximum number of undo steps
  operationHistory.setLimit(
      operationSupport.getUndoContext(), 10);

  try {
    // Execute the operation and add it to the history
    return operationHistory.execute(op, monitor, info);
  } catch (ExecutionException e) {
    return new Status(IStatus.ERROR, PLUGIN_ID, NLS.bind(
      "Cannot execute operation {0}", op.getLabel()), e);
  }
}
```

Listing 104: Method for the execution of the `IUndoableOperation` using the operation history

Location

To reduce duplicate code and use synergies, we decided to locate these methods in a utility bundle. So it's easier to use them in the other bundles. Additionally we tried to implement them as flexible as possible to allow a wide usage spectrum.

Customization

As an improvement of the command execution, it's possible to define several undo contexts using the interface `IUndoContext`. They are used to tag the operations as being applicable to a certain context. They are mainly used to filter the operation history for undo and redo so that only operations for a given undo context are shown when the application is presenting them.

We extended our implementation of the command execution to allow the user passing an own undo context. But we didn't use different contexts, because our applications were not very large and such a context handling would be an overhead.

6.3.10.1.2 Job & JobManager

Concept

As a further improvement we implemented jobs to allow an asynchronous execution of the command objects. Through the decoupling of the command creation and its execution it's possible to allow a faster reaction of user interactions. This is especially interesting for long-lasting operations, which would block the user interface. These and further advantages were the reason for our decision.

Implementation

We also decided to implement our job and further job classes in a generic manner. Together with a central location in the utility bundle, this allowed us to use

them for different operations in several other bundles. So we could reduce duplicate code.

The job

For their implementation itself RCP provides the abstract class `Job`, which can be used. It forces you to override the method `run` and a constructor passing the name of the job.

We decided to encapsulate the instantiation of the job objects in factory methods *[GoF1995]*. The reason for this decision is the possibility to do in the same step also configurations and to schedule it. Otherwise these steps had to be done in the classes using the job. This would lead to duplicate code, what we tried to avoid as much as possible.

Configuration

The settings we decided to configure in these factory methods are the priority and the scheduling rule. The priority affects how the job is scheduled while it is waiting to be run.

Different the scheduling rule. The Javadoc of the `ISchedulingRule`, the base interface, describes their purpose as follows: *"The scheduling rules are used by the jobs to indicate when they need exclusing access to a resource. The job manager guarantees that no two jobs with conflicting scheduling rules will run concurrently."*

We decided to define a default rule for the scheduling of the job, but to allow also passing an individual one if necessary. This gives the most flexibility for the later use in further bundles.

Execution of the command

In the method `run` we implemented the invocation of the method for the command object execution, like described above. Additionally we decided to allow the execution of the command object in the SWT thread. The reason for our consideration is the fact that command objects can interoperate directly with the user interface. But manipulations on them are only allowed in the SWT thread. Therefore we can reduce duplicate code, which would be necessary in the all affected user interface classes.

The **Listing 105** below shows an example implementation of such a job class with the constructor, the `run` method and several factory methods.

```
public class OperationJob extends Job {

  private static ISchedulingRule rule = new ISchedulingRule() {
    public boolean contains(ISchedulingRule rule) {
      return rule == this;
    }

    public boolean isConflicting(ISchedulingRule rule) {
      return rule == this;
    }
  };

  public static OperationJob executeOperation(
        IUndoableOperation operation, IAdaptable info) {
    OperationJob job = new OperationJob(operation, info, false);

    job.setPriority(INTERACTIVE);
    job.setRule(rule);
    job.schedule();
```

```
    return job;
  }

  ...

  private IUndoableOperation operation;
  private IAdaptable info;
  private boolean joinSwtThread;

  private OperationJob(IUndoableOperation operation,
        IAdaptable info, boolean joinSwtThread) {
    super(operation.getLabel());
    this.operation = operation;
    this.joinSwtThread = joinSwtThread;
    this.info = info;
  }

  @Override
  protected IStatus run(final IProgressMonitor monitor) {
    IStatus status;

    if (joinSwtThread) {
      Shell shell = (Shell) info.getAdapter(Shell.class);
      if (shell != null) {
        shell.getDisplay().syncExec(new Runnable() {
          public void run() {
            status = new OperationUtil().performOperation(
                operation, monitor, info);
          }
        });
        return status;
      }
    }

    return new OperationUtil().performOperation(
        operation, monitor, info);
  }
}
```

Listing 105: Implementation of a job for the asynchronously execution of command objects

Job manager

With the `JobManager` itself we had no direct contact. It is mainly used after the invocation of the schedule method. But these steps are implemented in the abstract class `Job`.

6.3.10.1.3 User Interface integration

Description

The previous implementations are only focused on the handling in the application itself. The undo and redo functionality cannot be used, because the user interface integration is missing. Therefore we had to extend our user interface to allow their use.

Action bar integration

An obvious extension is in the action bars. We added the commands, which are already provided by RCP, to the menu and toolbars like described in chapter 6.3.5. We decided to do it using the extension points, because it's more flexible and generic compared with programmatic implementation of them.

Link with the views

Typically this is enough and no further integration has to be done, because the global undo context is used generally. But in some cases it's required to implement the link with the views by hand, especially where another undo context is used.

In these cases we had to create new `UndoActionHandler` and `RedoActionHandler` instances with the corresponding undo context in the views. Additionally we had to register them in the action bars, which are typically accessible through corresponding methods. The **Listing 106** shows such an example.

```
// Undo/Redo
private IUndoContext undoContext
    = IOperationHistory.GLOBAL_UNDO_CONTEXT;
private UndoActionHandler undoAction;
private RedoActionHandler redoAction;
```

```
// Link with the Undo/Redo Actions
undoAction = new UndoActionHandler(this.getSite(), undoContext);
redoAction = new RedoActionHandler(this.getSite(), undoContext);

IActionBars actionBars = getViewSite().getActionBars();

actionBars.setGlobalActionHandler(
    ActionFactory.UNDO.getId(), undoAction);
actionBars.setGlobalActionHandler(
    ActionFactory.REDO.getId(), redoAction);
```

Listing 106: Implementation of the undo/redo link in a view using the global undo context

6.3.11 Common Navigator Framework

Overview

Depending on the purpose of the application, a navigator for managing projects or the organization of other parts is a useful feature. Like in the Windows Explorer the user has a good overview about the elements and can easily navigate through them.

Eclipse RCP provides the developers with a whole framework, the *Common Navigator Framework*, for the configuration and implementation of such navigators in a straightforward way. Compared with the features and frameworks out of RCP it provides a lot of default implementations. This allows its configuration simply through extension points. Therefore additional implementations can be reduced to a minimum.

6.3.11.1 Our Solution

Implementation

In our solution we also implemented a navigator component for the simple organization of projects containing files with the content for the graphical and text editors. We decided to integrate a navigator, because it allows the user an easier management of its files organized in projects. Otherwise the file management has to be done manually.

For the implementation we followed mainly the descriptions and instructions given by Eclipse's Platform Plug-in Developer Guide *[CNFProgGuide]* and

[Elder06].

View

In a first step we had to integrate the navigator view in our application.

Firstly, we decided to do this using the extension point `org.eclipse.ui.views`, like shown in **Listing 107**, because it is the most compact way. Otherwise we had to create a new class inheriting from `CommonNavigator`, which leads automatically to programmatic configurations, which we tried to prevent as much as possible.

```
<extension point="org.eclipse.ui.views">
  <view
    class="org.eclipse.ui.navigator.CommonNavigator"
    id="org.herasaf.tutorial.personlocator.ui.Navigator"
    name="Navigator"
    restorable="true">
  </view>
</extension>
```

Listing 107: Definition of the navigator using the `org.eclipse.ui.views` extension point

After further research we found a more compact way for the view integration, which has the advantage, that all default configurations are already done. Instead of defining an own view we used the already existing `org.eclipse.ui.navigator.ProjectExplorer`, which is provided by RCP. We only had to add it to our perspective, as shown in **Listing 108** below.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.personlocator.ui.perspective">
    ...
    <view
      closeable="true"
      id="org.eclipse.ui.navigator.ProjectExplorer"
      minimized="false"
      moveable="true"
      ratio="0.25"
      relationship="left"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      standalone="true"
      visible="true">
    </view>
    <viewShortcut
      id="org.eclipse.ui.navigator.ProjectExplorer">
    </viewShortcut>
    ...
  </perspectiveExtension>
</extension>
```

Listing 108: Addition of the `org.eclipse.ui.navigator.ProjectExplorer` to a perspective

Configuration

After the addition of the navigator we had to configure it using the extension point `org.eclipse.ui.navigator.viewer`. It tells the framework, which view part a Common Navigator is, what's the structure of the popup menu and

further more.

Firstly in our solutions we simply added the default configurations for the action and context binding, like shown in **Listing 109** below.

The action binding with the pattern `org.eclipse.ui.navigator.resources.*` is responsible to show the popup menu if the user does a right-click.

Different the content bindings which define the display of the projects and files, the integration of the filter functionality and the link feature for synchronizing the selection of the editor element and the tree element. Further the working set feature, which allows the user to group its projects.

```
<extension point="org.eclipse.ui.navigator.viewer">
  <viewer
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
  </viewer>
  <viewerActionBinding
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
    <includes>
      <actionExtension
        pattern="org.eclipse.ui.navigator.resources.*">
      </actionExtension>
    </includes>
  </viewerActionBinding>
  <viewerContentBinding
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
    <includes>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resourceContent">
      </contentExtension>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.filters.*">
      </contentExtension>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.linkHelper">
      </contentExtension>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.workingSets">
      </contentExtension>
    </includes>
  </viewerContentBinding>
</extension>
```

Listing 109: Configuration of the navigator using the extension point `org.eclipse.ui.navigator.viewer`

Through the further research, we already mentioned above, we found the already defined view `org.eclipse.ui.navigator.ProjectExplorer`, which has done all these default configurations by default. So we removed our own default configurations and had a navigator with the same default behavior.

Extension

Of course it is also possible to define own action providers, filters, wizards and viewer contents. For this purpose Eclipse RCP provides the extension point `org.eclipse.ui.navigator.navigatorContent`.

Because of the limited functionality of our applications, we decided not to do such extensions for our navigator.

6.3.12 Problems View

Overview

Notifying the user about errors, problems, warnings and further more is a very important feature, especially in editors where their content is checked by a parser, complier or another component. Since the results of such components are typically bigger than a simple message, their visualization with popups or other dialogs is insufficient. Additionally the message would be shown only as long as the dialog is open.

To close this gap a further component is required, which is able to visualize the whole information of such results. Eclipse RCP provides for this purpose the *Problems View*, which is widely used in the Eclipse IDE to show errors and warnings in the source code, like shown in **Figure 23** below. Additionally to the extended visualization it provides also the functionalities for grouping and sorting the entries.

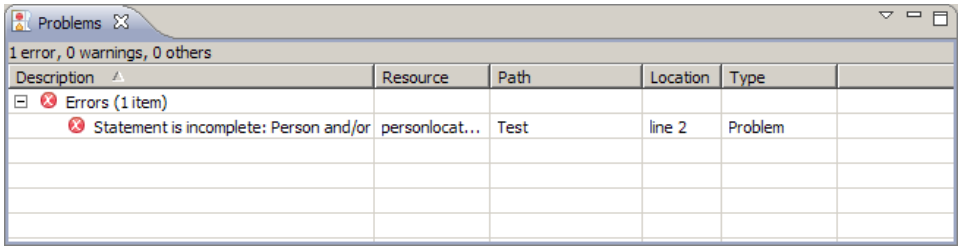

Figure 23: *Problems View* component for the visualization of the errors, problems, warnings and much more

6.3.12.1 Our Solution

Implementation

We also decided to integrate the *Problems View* in our application to display the errors and problems, which occurred during the parsing of the files contained in the projects managed by the navigator component. In our opinion it's the best way to show them.

Its implementation is very compact, because Eclipse RCP provides a wide range of classes and extension points which can be used without changes.

Problems View

To integrate the *Problems View* in our user interface, we only had to extend our perspective definition, like shown in **Listing 110** below. For the additional view we could use the id `org.eclipse.ui.views.ProblemView`. It is available after the addition of the `org.eclipse.ui.ide` plug-in to the *Required Plug-ins*.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.personlocator.ui.perspective">
    ...
    <view
      closeable="true"
      id="org.eclipse.ui.views.ProblemView"
      minimized="false"
      moveable="true"
```

```
      ratio="0.66f"
      relationship="bottom"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      visible="true">
    </view>
    ...
  </perspectiveExtension>
</extension>
```

Listing 110: Addition of the *Problems View* to the perspective

Marker definition

But this results only in an empty *Problems View* without columns. Therefore we had to configure it in a further step using the extension point `org.eclipse.ui.ide.markerSupport`. It allows the definition of attribute groups, marker fields and groups, marker types and problem filters.

We used the `markerContentGenerator` to configure the columns for the *Problems View*, like shown in **Listing 111** below. The fields we used for this purpose were already defined by Eclipse RCP. They cover the most common ones, like e.g. the problem description, the affected file and its path, the location in the file itself as well as the problem type.

```
<extension point="org.eclipse.ui.ide.markerSupport">
  <markerContentGenerator
    id="org.eclipse.ui.ide.problemsGenerator"
    name="Problems Generator">
    <markerFieldReference
      id="org.eclipse.ui.ide.severityAndDescriptionField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.resourceField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.pathField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.locationField">
    </markerFieldReference>
    <markerTypeReference
      id="org.eclipse.core.resources.problemmarker">
    </markerTypeReference>
  </markerContentGenerator>
</extension>
```

Listing 111: Definition of the markers for the *Problems View* using the extension point `org.eclipse.ui.ide.markerSupport`

Editor integration

So this is only the visualization part for the *Problems View*. Now we had to create and update the markers which should be displayed. This can be done for example in the text editor if its content is parser or compiled and returns appropriate results.

We decided to implement the marker update not in the text editor class itself, but to outsource it in a class with a common interface, like shown in **Listing 112**. The reason for this decision is the possibility to exchange the implementations easily through substitutes. This allows the adaption of the marker update algorithm without changes in the text editor class itself. Otherwise this algorithm

would be coupled directly with the text editor. This would work also fine but is bad design, because it reduces the cohesion of the class and leads to duplicate code if such an algorithm is used more than once.

```
public interface IMarkerHandler {

  public void updateMarkers(ITextEditor editor,
      List<ParseException> parseExceptions);
}
```

Listing 112: Definition of a common interface for the marker update

In our concrete implementation we had to get the `IFile` element out of the `IFileEditorInput`, because only on `IResource` objects like a file the markers can be created. So in a first step we removed all the existing markers from the resource object to prevent duplicate entries. Afterwards we created new ones based on the `ParseException`s we passed the marker handler as the result objects from our parser. It provides all the required information for the marker elements.

The creation of the `IMarker` elements is the only thing we had to do. The remaining steps are handled by Eclipse RCP itself, which shows the new entries in the *Problems View*.

The following code snippet in **Listing 113** shows such an implementation of a marker handler invoked from an editor passing a list with `ParseException`s for the marker creation.

```
public class DefaultMarkerHandler implements IMarkerHandler {

  protected void clearMarkers(IFile file) {
    try {
      IMarker[] markers = file.findMarkers(IMarker.PROBLEM, true,
          IResource.DEPTH_INFINITE);
      for (IMarker marker : markers) {
        marker.delete();
      }
    } catch (CoreException e) {
      e.printStackTrace();
    }
  }

  private int getDocumentPosition(ITextEditor editor,
      int lineNumber, int position) {

    int documentPosition = -1;

    try {
      IDocument document = editor.getDocumentProvider().
          getDocument(editor.getEditorInput());
      documentPosition = document.getLineOffset(lineNumber);
      documentPosition += position;
    } catch (BadLocationException e) {
      // do nothing
    }

    return documentPosition;
  }

  public void updateMarkers(ITextEditor editor,
```

```
      List<ParseException> parseExceptions) {

    if (editor.getEditorInput() instanceof IFileEditorInput) {
      IFileEditorInput input =
          (IFileEditorInput) editor.getEditorInput();
      IFile file = input.getFile();

      clearMarkers(file);

      for (ParseException parseExc : parseExceptions) {
        try {
          IMarker marker = file.createMarker(IMarker.PROBLEM);

          marker.setAttribute(IMarker.SEVERITY,
              IMarker.SEVERITY_ERROR);
          marker.setAttribute(IMarker.MESSAGE,
              parseExc.getMessage());
          marker.setAttribute(IMarker.LINE_NUMBER, parseExc
              .getLineNumber());
          marker.setAttribute(IMarker.CHAR_START,
              getDocumentPosition(editor, parseExc
                  .getLineNumber(), parseExc
                  .getStartPosition()));
          marker.setAttribute(IMarker.CHAR_END,
              getDocumentPosition(
                  editor, parseExc.getLineNumber(), parseExc
                  .getEndPosition()));
        } catch (CoreException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

Listing 113: Implementation of a marker update algorithm

In the text editor class itself, only an instance of the `IMarkerHandler` must be available for invoking it after the parser execution. Such an instance can be injected for example with Spring or created simply with the `new` operator.

6.3.12.2 Problems

Missing plug-in

After the configuration and implementation of the *Problems View* feature, we tried to start the application. Unfortunately it failed with an error, occurred through a missing plug-in. We could solve the problem easily through the addition of the plug-in `org.eclipse.ui.views.properties.tabbed` which seems to be used as the base of the *Problems View*.

6.3.13 Multilanguage support

Overview

The support of multiple languages is an important feature in cases where an application is used in different regions. Thus, the addition of multilanguage support raises the flexibility and usability of the application.

In this chapter we describe our solution concerning the implementation of the

multilanguage support.

String externalization mechanism

For this issue the Eclipse framework provides a string externalization mechanism. It supports the developer with the automatic lookup for loose strings in the projects, the generation of a property file containing the messages and a utility class to access them.

6.3.13.1 Our solution

Concept

Generally, the string externalization mechanism recognizes all loose strings inside of java classes. In our solution we only externalize strings which are related to the presentation of the application. For strings which are used for other concerns like the application configuration, we recommend creating an own configuration file.

Language bundle

For the implementation of the multilanguage support we use an own bundle. The language bundle holds a utility class for the message access as well as a property file containing all the messages. It gets referenced by other bundles requiring the text messages for the description of the UI. The `Messages` class provides a method `getString` for the message access. In conjunction with a key, this method returns the appropriate message value.

6.3.14 Wiring with Spring

Overview

The well-known Spring Framework is a very useful tool for wiring applications at runtime, to mention only one of its features. Because of that, it's possible to implement the applications strongly against interfaces and abstract classes because the instances of the concrete classes will be injected during the startup.

This feature is very valuable for the development and testing of an application. For this reason it is very advantageous to utilize it also in Eclipse RCP applications.

Spring Framework Utilities

Wiring an RCP application was not possible so far, because RCP follows a different concept with its extension points.

Through the implementation of the *Spring Framework Utilities* by Martin Lippert *[Lippert08]* this gap is closed. This means the utility cases in between the extension point and the Spring Framework. The definition of the class which should be instantiated and invoked through the extension point must be replaced with the `org.eclipse.springframework.util.SpringExtensionFactory`, followed by a colon and the name of the bean out of the application context.

```
org.eclipse.springframework.util.SpringExtensionFactory:editor
```

Listing 114: Class definition for an extension point using the Spring Framework Utilities

The `SpringExtensionFactory` looks up the bean with the passed name in the application context and returns it. The extension point mechanism itself

doesn't notice anything from this.

6.3.14.1 Our Solution

Implementation

During the implementation of our applications, we decided to use the *Spring Framework Utilities*. Otherwise we had to implement an own way to get the beans out of Spring's application context. Since such a solution would lead to additional dependencies it was no option for us.

Component wiring

The wiring of our components was now possible like in a conventional application. For this purpose we defined in the application context the required beans and passed them either as constructor parameters or injected them as properties.

This allowed us a more flexible implementation of the applications against interfaces and abstract classes.

6.3.14.2 Problems

During the use of the *Spring Framework Utilities* and consequently the Spring Framework itself, we had some problems to solve.

Application wiring not possible

The first one occurred during the definition of the application. We tried to use the `SpringExtensionFactory` in the extension point `org.eclipse.core.runtime.applications`. Unfortunately this lead to exceptions, because the bundle could not be found.

Although we increased the start priority of the Spring bundles including the *Spring Framework Utilities*, we could not solve this problem. It seemed that the wiring of the application skeleton is not possible, because not all required resources were started until then. So we decided to set up the application skeleton without Spring.

Component does not open (Not available service)

Different were the problems with the components which didn't open. Although its wiring was correct and worked fine it was not opened. After some research and trials, we found out, the problem occurs if an OSGi service is referenced which isn't registered yet. For services, which have the cardinality *0..1* or *0..N* this is no problem, only for the mandatory ones.

We could solve this problem on the one hand through the definition of the start order, but with the disadvantage to suppose a specific order. Additionally this would break the idea of the OSGi framework, which allows registering and deregistering services at any time.

So we decided to change the cardinality. But this leads to additional efforts in the class using it. We had to check if the reference exists before we used them. For this purpose it may be required to cache data locally until the service is available.

Alternatively also the definition of a lookup timeout is possible after which an exception is thrown, which can be shown.

Constructor parameter

A further challenge occurred through the classes provided by RCP. Some of them require resources, which must be passed in the constructor. Generally this is no problem, only in cases where these resources are part of the components where they should be injected. Spring has no access to them.

To solve this problem we had to implement an indirection between the class which should be instantiated and the other one where it should be injected. The indirection layer is mainly a factory creating the class with the passed resources.

Listing 115 shows an example of such an indirection layer.

```
public interface ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
        EditPartViewer viewer, ActionRegistry registry);
}


public class DefaultContextMenuProviderFactory implements
      ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
        EditPartViewer viewer, ActionRegistry registry) {
    return new DefaultContextMenuProvider(viewer, registry);
  }
}
```

Listing 115: Example of a indirection layer to allow a proper Spring wiring

6.4 Control Layer

Overview

This chapter describes the control layer in our solution.

Intention

The main intention of the control layer is to take different responsibilities in order to increase the flexibility and reusability of the application. By keeping the RCP dependencies low a detachment of the UI layer is achieved.

The control layer merely has dependencies to the core project, which includes the domain model representations, and refers to the service layer. Without a control layer the responsibility in the UI components would have been too high, which leads to low cohesion and high coupling.

Responsibility

As mentioned before, the control layer carries dependencies to the core project and to the services in the service layer. Besides of the advantages concerning software-engineering principles, this layer also has the following responsibilities:

- Communicate with the services in the service layer
- Prepare the data for the use in UI components
- Parse the domain objects into UI specific ones and vice versa
- Carry the UI specific domain objects (like e.g. for the graphical editor)
- Handle the notification on domain model changes

Service reference

For the construction of the service references Spring DM is used. The wiring is configured in the UI main project. A detailed description of this topic can be found in chapter 6.3.14.

Notification

In case of domain model changes, the UI components have to be notified in order to show an actual representation of the data. In our solution we use the Whiteboard Pattern *[Kriens04]* for this issue. Thus, there are two types of services in our solution. The ones which provide domain functionality and others which represent change listeners. A listener class implements a specific listener interface and is registered as an OSGi service in the OSGi service registry. Services processing domain functionality are now able to source listener services and send a notification to them on changes.

Parsers

Parsers are responsible to transform UI specific model data into domain model data and vice versa. They work closely together with UI components like e.g. the text editor.

In our solution the text editor and as well as the graphical editors implemented their own parsers in the appropriate control layers.

6.4.1 Problems

Service availability problems

Depending on how the bundles are installed on startup of the application it is possible that a service is not yet available at the time of an action. This issue can be solved even through the reconfigure of the startup configuration in the *config.ini* descriptor or by changing the cardinality in the application context and the necessary check if the service is null when using it.

6.5 OSGi Service Layer

Overview

Having a look into the OSGi principles, the service-orientation is intended to offer flexibility by loose coupling. Also, by offering a publish-find-bind service model, existing assets can be integrated transparently.

In order to benefit from these advantages, the main intention of the OSGi Service Layer is to offer service implementations for different service interfaces, register them in the service registry and make them available for anyone in the OSGi environment. Through this abstraction, it's also possible to easily provide different implementations for a service interface, as long as the implementation complies with the interface specifications.

6.5.1 Bundle distribution

Intention

The issue of how to distribute the service bundles is very important. Depending on how you organize your service interfaces and service implementations, the overhead when loading your bundles and the coupling between them can be adjusted.

In this chapter we will show you how we solved these problems, which alterna-

tives were possible and which one of them we chose.

6.5.1.1 Our Solution

Distribution

In our solution we've created a services bundle including all the service interfaces of the application. For the concrete service implementations we use separate bundles.

Taking a calculator application as example, we'd have a *Services* bundle containing all service interfaces describing the services provided in the application. This services bundle could e.g. contain a service, which calculates a result out of a string including numbers and mathematical operations. On the other side we'd have a bundle for each service definition implementing the concrete functionality.

Dependencies

In order to have access to the required service interfaces, an implementation bundle has to import the appropriate package (*Import-Package*) in its manifest file.

To ensure that you only import the interfaces you want to implement and to avoid importing service interfaces you don't need, the interfaces are placed in separate packages inside of the services bundle.

Bundle Diagram

Referring to the calculator application example mentioned above, the bundle diagram would look like in the figure below.

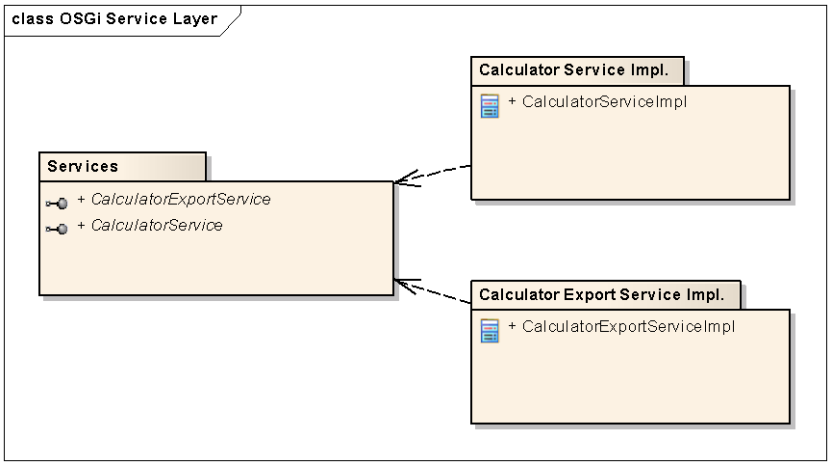

Figure 24: Distribution and dependencies of the bundles in the calculator example

6.5.1.2 Alternatives

Overview

To solve the problem of the bundle distribution, several alternatives were possible. To be exact, the main concern of the problem is how to manage and distribute the service interfaces and the associated concrete implementations into bundles. For the review of the alternatives the following aspects were kept in

mind:

- Effort for the service client to bind and use a service
- Cohesion of a bundle
- Coupling between the involved bundles
- Versioning issues in case of changes
- Addition of new bundles and functionality
- Difficulty to configure and manage the dependencies
- Effort to setup the application environment

Please note, that there are many more possible combinations for alternatives. We will describe only the combinations that came into consideration for the solution of this problem.

In this chapter, these alternatives are described including the advantages and disadvantages. After the description of the alternatives the decision gives the reasons for our design choices.

Alternatives

Alternative 1: One bundle per interface, one per implementation

The approach of this alternative is to have one bundle per service interface as well as one bundle per service implementation.

Concerning the dependencies, the service client references multiple service bundles in order to have access to the describing service interfaces. Additionally, each service implementation bundle references the corresponding service interface bundle to be able to create a concrete implementation of it.

Taking the calculator application as an example again, **Figure 25** shows the distribution and the dependencies.

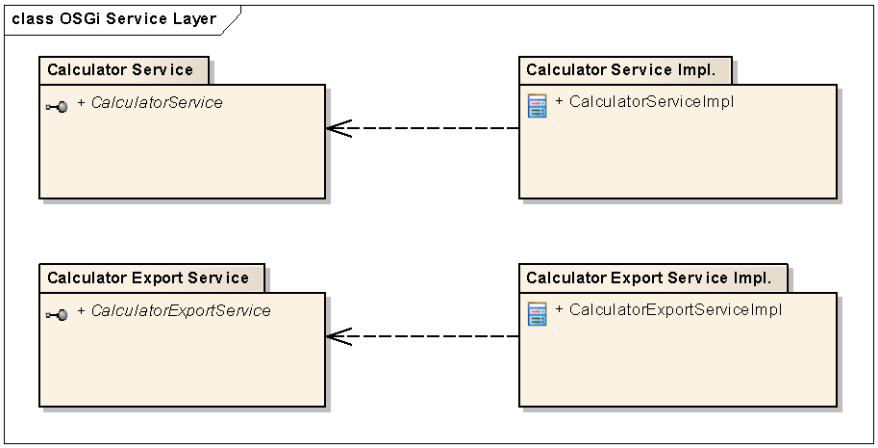

Figure 25: "One bundle per interface, one per implementation" bundle distribution

Advantage	Disadvantage
• Interface definitions and concrete implementations are separated. New implementations and/or changes can be ap-	• Complex dependency model between bundles. Having a large amount of services requires high configuration and management ef-

plied easily. • Service implementations are located in own bundles. Changes on them will only affect the concerned bundle. • The service client imports only interfaces. Loose coupling between service interfaces and concrete implementations. • Full separation of concerns possible and high cohesion. Install only really used and required services for the application. • Individual versioning of service interfaces and service implementations possible.	fort. • Implies a large total amount of bundles. Difficult to keep the overview and to set up the application environment. • Low granularity of the grouping. In practice services can be grouped by their main concerns. • Individual versioning could be a problem because in practice the implementation changes more often than the interface definitions. In such a case the effort to manage the dependencies would grow massively.

Alternative 2: One bundle for all interfaces, one for all implementations

This alternative specifies a bundle including all service interfaces for the application, whereas another bundle contains all the associated service implementations.

There is a dependency from the implementation bundle to the service interface bundle, in order to have access to the service interfaces. The service client simply has to reference the service interfaces bundle and thereby has access to all the provided service interfaces in the application.

In large systems, where a large amount of services is existent, a separation of service interfaces and service implementations in different groups can be considered.

Figure 26 shows the distribution and the dependencies in case of the calculator application example.

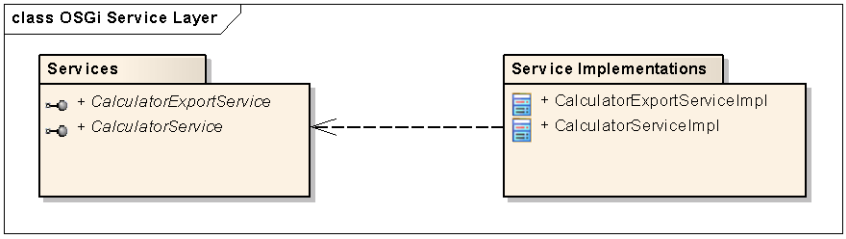

Figure 26: "One bundle for all interfaces, one for all implementations" bundle distribution

Advantage	Disadvantage
• Interface definitions and concrete implementations are separated. New implementations and/or changes can be applied easily. • Simple dependency model.	• Even if a service client does not need all available service interfaces, all of them will be involved with the reference to the bundle. • In order to make a specific service implementation available, all of the

Marginal effort needed to configure and provide the required service bundles.

- Small total amount of bundles for the services. It is easier to keep the overview and set up the environment.
- Services can be grouped by their main concerns. By choosing an appropriate granularity level, the coupling between the service client bundles and the service bundles can be loosened.

service implementations in the bundle will be involved.

- Without a grouping, the cohesion of the bundles can drop because of the concentration in only one bundle.
- On addition, update or change of a service implementation, all implementations will be involved, because they are located in the same bundle.
- On addition, update or change of a service interface, all interfaces will be involved, because they are located in the same bundle.
- Individual versioning of service interfaces and/or service implementations is not possible.

Alternative 3: One bundle for all interfaces, one per implementation

The approach of this alternative is a combination of the first two alternatives. There is a bundle containing all provided service interfaces in the application and an own separate bundle for each service implementation.

For the service lookup, the service client simply references the service interfaces bundle. Concrete service implementations all have a dependency to the service interfaces bundle.

A separation of service interfaces into different groups is also considered in the case where a large amount of services are available.

Figure 27 shows the distribution and the dependencies in case of the calculator application example.

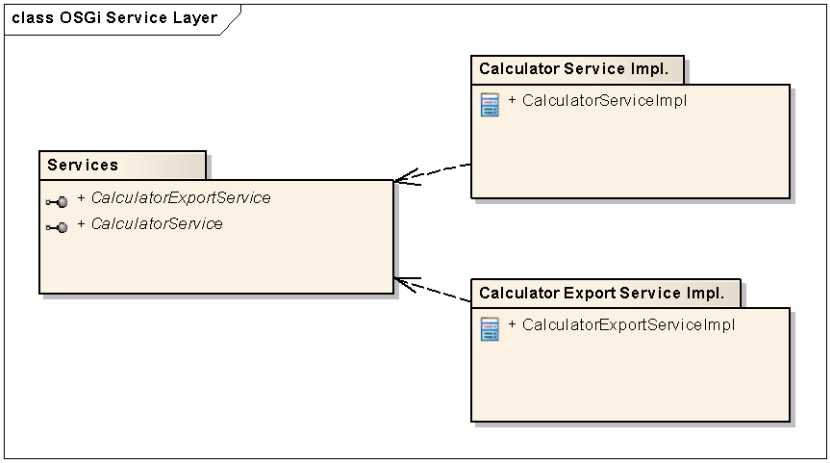

Figure 27: "One bundle for all interfaces, one per implementation" bundle distribution

Advantage	Disadvantage
• Interface definitions and concrete implementations are separated. New implementations and/or changes can be applied easily. • Service implementations are located in own bundles. Changes on them will only affect the concerned bundle. • Service implementations can be integrated independently from other service implementation bundles. • Service implementations can be versioned individually whereas the service interfaces are versioned as a group. • Low effort for the service client to get access to the service interfaces. It simply can reference the service interfaces bundle. • By grouping the service interfaces by their main concerns, the cohesion of the bundles can be increased.	• On addition, update or change of a service interface, all interfaces will be involved, because they are located in the same bundle. • Even if a service client does not need all available service interfaces, all of them will be involved with the reference to the bundle. • Without a grouping, the cohesion of the bundles can drop because of the concentration in only one bundle.

Alternative 4: All in one bundle

In this alternative, all the service interfaces including the associated implementations are located in only one bundle. The service interfaces are published (*Export-Package*) whereas the concrete implementations are not (*Private-Package*).

The service client references this one bundle and uses the exported service interfaces to lookup the service in the service registry. There are no other dependencies to other bundles, as in the other alternatives.

The grouping of services by their concerns is considered here, too. Such a separation could be the creation of multiple bundles, all of them following the schema of containing the interfaces and the implementation in one bundle.

Figure 28 shows the distribution and the dependencies in case of the calculator application example.

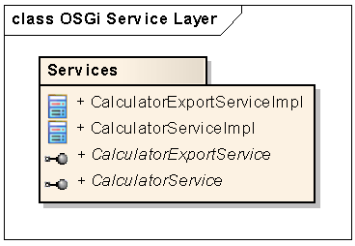

Figure 28: "All in one bundle" bundle distribution

Advantage	Disadvantage
• Simple dependency model. The effort needed to configure and provide the required service bundles is low. • By grouping the service interfaces and their implementations, the cohesion of the bundles can be increased. • Small total amount of bundles for the services. It is easier to keep the overview and set up the environment.	• Even if a service client does not need all available service interfaces and implementations, all of them will be involved with the reference to the bundle. • In order to make a specific service implementation available, all of the service interfaces and implementations in the bundle will be involved. • Without a grouping, the cohesion of the bundles can drop massively because of the concentration of interfaces and implementation in only one bundle. • Interface definitions and concrete implementations are all in one bundle. New implementations and/or changes cannot be applied easily. • On additions, updates or changes, all interfaces and implementations will be involved, because they are located in the same bundle. • Individual versioning of service interfaces and/or service implementations is not possible.

Alternative 5: As *Alternative 1*, but with default implementation

The approach of this alternative is the same as in *Alternative 1*. The difference is that in this case a default implementation of the service interfaces contained in the service interfaces bundle is provided.

Concerning the dependencies, they are the same as in *Alternative 1*. The service client references multiple service bundles in order to have access to the describing service interfaces. Each service implementation bundle references the corresponding service interface bundle to be able to create a concrete implementation of it.

Figure 29 shows the distribution and the dependencies in case of the calculator application example.

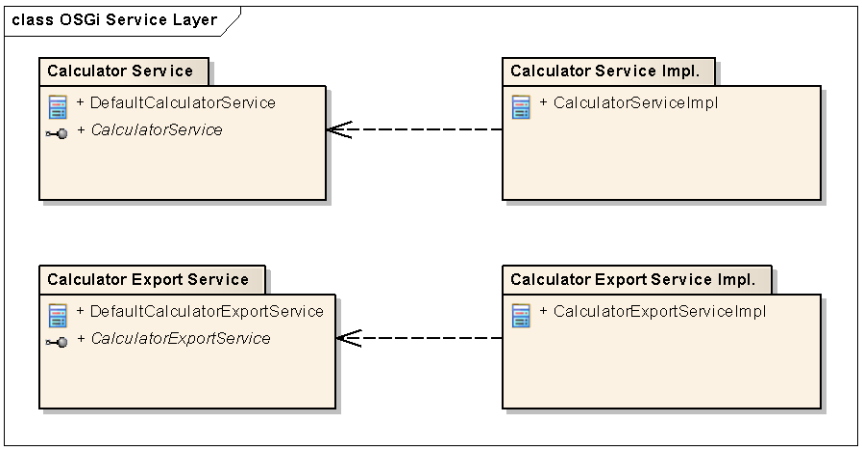

Figure 29: "As *Alternative 1*, but with default implementation" bundle distribution

Advantage	Disadvantage
• Interface definitions and concrete implementations are separated. New implementations and/or changes can be applied easily. • Service implementations are located in own bundles. Changes on them will only affect the concerned bundle. • The service client imports only interfaces. Loose coupling between service interfaces and concrete implementations. • Full separation of concerns possible and high cohesion. Install only really used and required services for the application. • Individual versioning of service interfaces and service implementations possible. • A concrete service implementation is provided by default. Assures the availability of a service if no other is defined and keeps the coupling and the total amount of bundles low by only defining a service implementation bundle when specific functionality is needed.	• Complex dependency model between bundles. Having a large amount of services requires high configuration and management effort. • Implies a large total amount of bundles. Difficult to keep the overview and to set up the application environment. • Low granularity of the grouping. In practice services can be grouped by their main concerns. • Individual versioning could be a problem because in practice the implementation changes more often than the interface definitions. In such a case the effort to manage the dependencies would grow massively. • Service ranking definitions have to be considered when "overwriting" the default service implementation with a specific one.

Decision

The decision, which alternative to choose, really depends on the scope of the application. But nonetheless it can be said, that there are better solutions which

help increasing the maintainability and portability of the whole buildup. In the case of this problem, the compromise stands between low management effort and high interchangeability.

Based on the advantages and disadvantages in the comparison above, we chose the alternative, which specifies a bundle including all the service interfaces and separate bundles for each service implementation (*Alternative 3*). This solution is a good compromise between having a large amount of bundles, which leads to complex dependencies and makes configurability very difficult, and still staying flexible enough to avoid interchangeability issues. Also, in practice, services can be grouped by their main concerns very easily and service clients mostly will have the need to directly import a bunch of bundles in order to be able to provide a holistic functionality in their applications. Looking at the other alternatives from this point of view, they either lead to unnecessary individuality and therefore to more complexity or too much simplicity.

6.5.2 Service registration, lookup and binding

Intention

Another important matter is the whole management and the use of services in the environment. That is the lookup, the registration and the binding of concrete services.

To be able to use services in the OSGi environment, service implementations have to be registered in the service registry. Bundles requiring a service will look it up there and try to get an instance in order to bind the service. Following this principle, a service client can get access to a service and is able to do the job it is intended to.

There are several ways of implementing the management and the use of services in the OSGi environment. In this chapter we will show you which alternatives are possible and which one of them we chose.

6.5.2.1 Our Solution

Spring Dynamic Modules

The Spring Dynamic Modules *[SpringDM]* make it possible to use the functionality of the Spring framework *[Spring]* inside of applications which can be deployed in the OSGi environment. Additionally, Spring DM provides various OSGi service management features.

In our solution we use Spring DM for the dependency wiring and for the OSGi service management.

Bundle distribution

Referring to the decision for the bundle distribution in the last chapter, the service interfaces and the implementations are divided in separate bundles. The service interfaces bundle includes all provided interfaces for the application whereas there is an own bundle for each service interface implementation.

The service interfaces bundle does not contain a Spring DM application context, because it simply is a container of interfaces.

Service interfaces bundle

Therefore, it exports the service interfaces by declaring them as *Export-Package* in its manifest file and gets referenced by other bundles requiring the service interfaces.

There is no Spring DM application context existent in the service interfaces bundle, because it only includes the interface definitions and does not contain nor know any concrete implementations.

Service implementation bundle

Different the service implementation bundle, which specifies a concrete service implementation. It has a dependency to the service interfaces bundle. To be exact, it imports the packages, which contain the required service interfaces, using *Import-Package* in its manifest file. The implementation class itself is defined as *Private-Package* and thus is not visible for other bundles.

Unlike in the service interfaces bundle, here we have a directory called *META-INF/spring/* in the bundle which contains two XML files describing the Spring DM application context. The file *bundleContext-osgi.xml* contains the service declaration whereas *bundleContext.xml* defines the required beans for the service registration.

Service registration

The service declaration and registration in the service registry happens in the service implementation bundle. Spring DM, assumed that it is correctly installed and activated in the OSGi execution environment, reads the *META-INF/spring/* directory and creates the application context out of the specification in the XML files. According to the *<osgi:service>* declaration in the *bundleContext-osgi.xml* file, Spring DM registers the specified bean as an OSGi service.

Service client bundle

The service-orientation of the OSGi environment decouples the service client bundle completely from any service implementation. There simply is a dependency to the service interface bundle in order to have access on the service interface and to be able to lookup and bind a service instance.

Concerning Spring DM, the service client bundle also contains the *META-INF/spring/* directory with its application context describing XML files. The file *bundleContext-osgi.xml* contains a service reference whereas *bundleContext.xml* defines the required beans for the dependency injection.

Service lookup and binding

At the time, when Spring DM constructs the application context, an adequate service with the specified attributes will be looked up and bound. From now on, the service is available and can by referenced by any bean.

Configuration Diagram

To show how the application context configuration looks like practically, we take the calculator application as example. **Figure 30** shows the concrete entries in the application context configuration and how the service reference is injected into a model class of the calculator user interface.

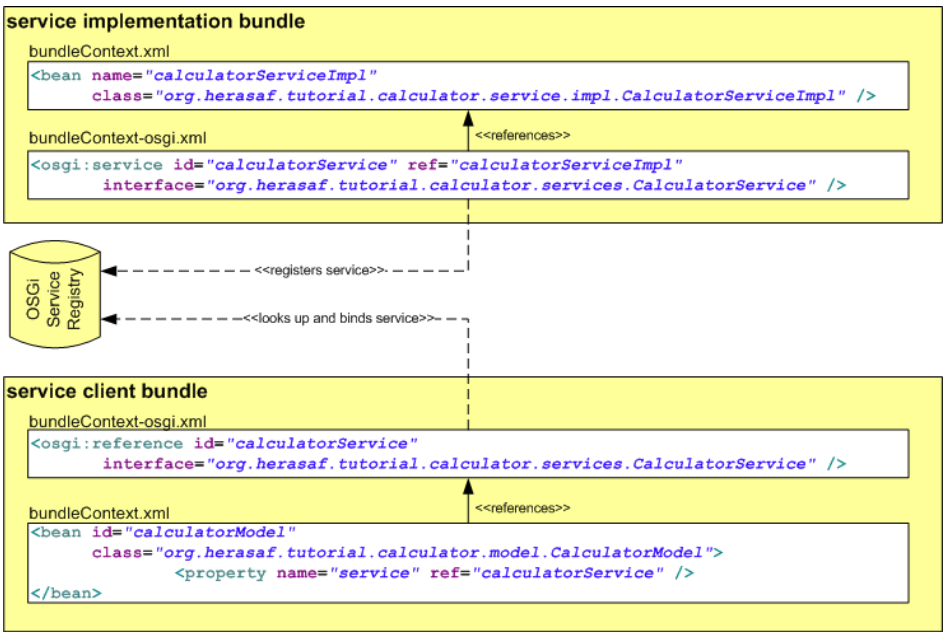

Figure 30: Spring DM application context configuration for the calculator example

Spring DM and RCP

Having a service client with a user interface, the service reference needs somehow to be integrated in the RCP environment. Spring DM cannot be used to directly inject RCP views and/or other RCP components in RCP applications. For more details we would like to refer to chapter 6.3.14.

6.5.2.2 Alternatives

Overview

As already mentioned before, there are several alternatives for the implementation of services in the OSGi environment. The main activities for the service management and their use are the registration into the service registry, the lookup and the service binding. Depending on the implementation and the technology used, these tasks can be different to handle. The alternatives were reviewed keeping the following aspects in mind:

- Implementation effort and complexity
- Need of updates in code on changes
- Service tracking

In this chapter these alternatives are described including the advantages and disadvantages. After the description of the alternatives the decision gives the reasons for our choice.

Please note that the description of the alternatives is not meant to be a full description of all the functionality provided by the different technologies. It is simply intended to give an idea of the possible solution.

Alternatives

Alternative 1: Use standard OSGi libraries

The OSGi environment provides standard libraries for the service implementation. The service implementation bundle defines the package containing the service implementations as a *Private-Package*. In its manifest file a *BundleActivator* has to be defined, which uses the *BundleContext* to register the ser-

vice implementations on startup.

On the client side, the service client bundle imports the required package including the service interface. Through the *BundleContext*, the service can be looked up in the service registry by using the service interface. OSGi also provides service tracking solutions, which can be used to react on lifecycle changes of the service, as e.g. on service binding.

Advantage	Disadvantage
• Full OSGi functionality and libraries available. • No extra libraries and bundles needed in order to implement services.	• Higher implementation effort and complexity for the service registration and lookup. • There are OSGi dependencies in the bundle. These are needed to implement the *Activator*. • The code in the *Activator* needs to be adjusted on changes (like e.g. on a new service implementation) • To ensure the availability a service tracker has to be implemented.

Alternative 2: Use Spring Dynamic Modules

Spring DM *[SpringDM]* offers the possibility to use Spring *[Spring]* dependency wiring for bundles that are deployed in the OSGi environment.

Having Spring DM installed and activated properly in the OSGi environment, it will check the bundles for an existing *META-INF/spring/* directory. In this directory all contained XML files will be read to build the application context.

The service implementation bundle simply has to configure the application context properly in order to have its service implementations registered in the service registry. All that has to be done is to create a bean for the service implementations and specify a service definition referencing the service implementation beans.

The difference on the client side is that instead of a service definition, a service reference has to be specified, which then gets referenced by various beans. Concerning service tracking functions, these can be directly configured in the application context inside of the <*osgi:reference*> tag.

Advantage	Disadvantage
• No *Activator* implementation needed in the bundle. Therefore there are no dependencies to OSGi. • No code updates needed in case of changes, like e.g. additional implementations, etc. • No need of service tracking. Automatic solution of availability issues through the use of	• Not all the functionality of OSGi provided. • Additional bundles and libraries needed in the OSGi environment.

service tracker functionality.

- Low implementation effort and easy configuration.

Decision

Our decision to use Spring DM for the implementation of services is based on the overbalance of advantages of this solution. Looking at the disadvantages, it's only a small price to pay for all the simplification and flexibility gained.

6.6 Utilities

Overview

This chapter covers the utility part of the developed reference architecture. This includes beside the concrete solutions also the comparisons of further alternatives and the decision for one of them.

6.6.1 Logging

Overview

Logging of specific application states, occurred errors or important workflow information helps the developer to understand what happened during the use of the application. This is really important for the analysis of errors or to find improvements. For this purpose a wide range of different logging frameworks exist, like Log for Java *[Log4J]* or the Simple Logging Framework for Java *[SLF4J]*.

But the use of such a framework in an OSGi based environment is a bigger challenge, because of several reasons. One of the most obvious one is certainly the divided classpaths in the OSGi environment. Additionally, the most of the logging frameworks aren't prepared for their use in an OSGi environment yet.

6.6.1.1 Approaches

Introduction

For the implementation and use of a logging functionality in an OSGi/RCP based application several solutions exist, like the Eclipse internal logging feature, but also well-known logging frameworks and the OSGi Log service. Each of them has its individual advantages and disadvantages.

In the following comparison we take a detailed look at the different concepts and consider them carefully against each other. This allows us to come to an appropriate solution.

Alternatives

Alternative 1: Use the Eclipse Logging

The Eclipse platform itself provides an own logging feature based on a status handling. It is automatically used when starting an RCP application, to log which plug-ins are loaded, which are missing and further more.

Advantage	Disadvantage
• It is well integrated in the Rich Client Platform. Therefore no	• For using the Eclipse logging framework, the base Eclipse plug-

additional configurations are required. • It is already used in the platform plug-ins itself why it is used automatically. • It writes the log files in the working directory of the application. Therefore no further configurations of the logging path are required. • It is exclusively designed for the use in an OSGi environment. Therefore no compatibility problems exist. • It is possible to extend the logging through the implementation of log listeners.	ins are required, even then a plug-in would be totally independent from Eclipse.

Alternative 2: Use a logging framework

An alternative solution is the use of a well-known logging framework, like Log4J and SLF4J. They provide a wide range of configuration possibilities to adapt its behavior and filter entries.

Advantage	Disadvantage
• They are well-known and widely used in the Java application development. • Through their adaptability they can be configured in detail, to filter log statements, write different log files and further more.	• Some of those frameworks are not prepared for the use in an OSGi based environment like in a RCP application. They must be converted before. • Their use in an OSGi environment is not well documented and seems therefore to be more in an experimental stage. • Some of them have special requirements, like for the classpath and further more. This can lead to an increased configuration effort.

Alternative 3: Use the OSGi Log Service

A further possibility provides the OSGi framework itself with its Log service. It is part of the framework specification itself and must be provided by its implementation.

The logging feature consists of two individual parts, like shown in the **Figure 31** and described by *[Wütherich2008]*. The first one is the logger itself which submits the log statements to the Log service. This stores them until the framework is shut down or the statements are read from the second part through the LogReader service. The log reader can then use the statements to write them in a log file, for reporting purposes or further more.

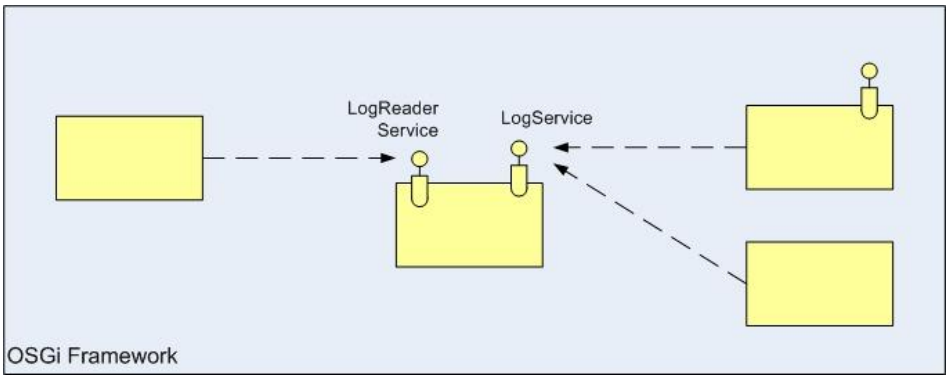

Figure 31: Schematic overview about the OSGi Log Service

Advantage	Disadvantage
• The Log service is a component of the OSGi framework. Therefore it exists in each framework which corresponds with the specification. • Through the open structure, it allows a flexible implementation of the Log and LogReader service users. It is possible to use the log statements for reporting purposes or to use different logging frameworks. • The configuration, implementation and use of this service are well documented, for example in *[Wütherich2008]*. • It is additionally possible to implement log listeners to get notifications about new log statements. • Through the user of the Log service component, no additional plug-ins and libraries are used. They are already contained in the platform.	• Compared with the other alternatives, the Log service component requires a higher initial effort. • The use of a service for logging is more complex than simply invoking a method of another logging framework.

Decision

Based on the comparison above, we decided to use the OSGi Log service for the logging purposes. The reason for this decision is on the one hand its openness which allows a great flexibility and on the other hand, through the simple configuration effort compared with the other logging frameworks. Additionally no further plug-ins and libraries are required, because they are all included by default in the target platform.

Different would be the use of the well-known logging frameworks like SLF4J or Log4J. Here are additional libraries required, which must be converted in a first step to bundles if they aren't yet. Additionally their requirements for the configu-

ration, the classpath handling and further more makes them very cumbersome to use. And all in all their use in an OSGi based environment isn't documented very well. That's why it seems to be still in an experimental stage.

Also different is the use of the Eclipse internal logging framework. It is part of Eclipse's Rich Client Platform, why its use is only possible if the Equinox implementation of the OSGi specification is used. Futher, it would be necessary to have dependencies to RCP plug-ins in each bundle and plug-in where the logging should used. This is especially for those a problem, which have no other Eclipse RCP dependencies. A solution for this problem would be the implementation of a bundle which encapsulates the access on the Eclipse internal logging.

But it would be possible to use the Eclipse internal logging as well as the other well-known logging frameworks in the LogReader to handle the incoming log statements, which proves the flexibility of this solution.

6.6.1.2 Our Solution

Our Solution

Based on the comparison in the chapter above, we decided in our solution to use the OSGi Log Service for logging purposes. It provides for adaptions and further extensions the greatest flexibility, compared with the other concepts and frameworks. So it is possible to use the log statements also for reporting purposes or to write them easily in a database without any changes in the remaining application.

Structure

For our solution we decided additionally to extend the given structure of the Log service through a log façade, as shown in the **Figure 32** below. The reason for this structural extension is mainly the decoupling of the application bundles from the log service. Therefore a log service instance is only used in the façade bundle. This reduces the lookup effort immensely which leads to an increased stability.

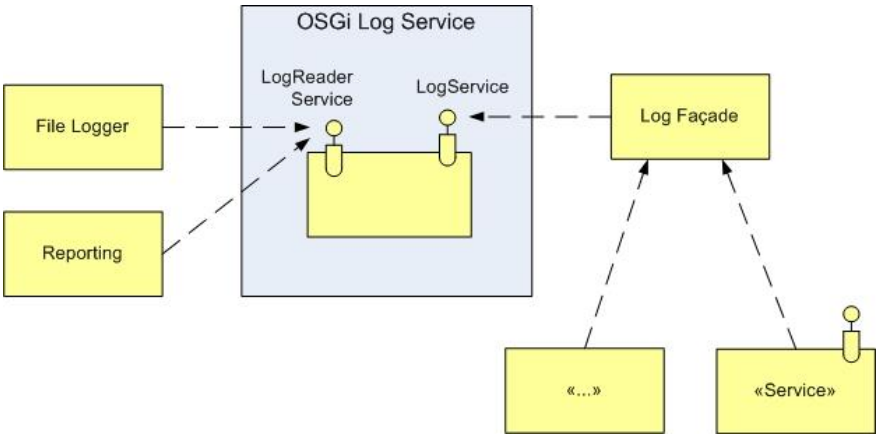

Figure 32: Schematic overview about the OSGi Log Service

Log façade

So we had to implement mainly two additional bundles, one containing the logging façade and another for the log reader. The first one allows creating log statements through simple method calls. It converts then the passed information

to a log statement, like shown in the code snippet of **Listing 116** below.

```
public void debug(String message) {
  logService.log(LogService.LOG_DEBUG, message);
}

public void debug(String message, Object arg) {
  logService.log(LogService.LOG_DEBUG,
      MessageFormat.format(message, arg));
}

public void debug(String message, Object[] args) {
  logService.log(LogService.LOG_DEBUG,
      MessageFormat.format(message, args));
}
```

Listing 116: Methods in the log façade to create a log statement using the passed parameters

Since the Log service is an OSGi component, it was required to create a new folder *OSGI-INF* with the file *component-description.xml*. This file contains the component description, like shown in **Listing 117** below. Otherwise the use of the Log service is not possible.

```
<?xml version="1.0"?>
<component name="logger">
  <implementation
      class="org.herasaf.tutorial.personlocator.log.Logger" />
  <reference name="logService"
      interface="org.osgi.service.log.LogService"
      bind="setLogService" unbind="unsetLogService" />
</component>
```

Listing 117: Component description of the Log service

Additionally we had to declare in the bundle manifest with the header `Service-Component`, where in the bundle the component description can be found.

```
Service-Component: OSGI-INF/component-description.xml
```

Listing 118: Additional header in the bundle manifest for the component declaration

LogReader

Very similar was also the implementation of the LogReader. We used the `LogReaderService` instead the `LogService` and registered us using the `LogListener`. Therefore we had the chance to get notified about new log statements and don't have to poll regularly.

Since the Whiteboard Pattern *[Kriens04]* was invented after the specification of the Log service, it is not possible to simply register a service. Instead, we had to register the log listener on the LogReader service, like shown in **Listing 119** below.

```
public class ConsoleLogListener implements LogListener {

  protected LogReaderService logReaderService;
```

```
  public void setLogReaderService(LogReaderService logReader) {
    this.logReaderService = logReader;
  }

  public void unsetLogReaderService(LogReaderService logReader) {
    this.logReaderService = null;
  }

  protected void activate(ComponentContext context) {
    logReaderService.addLogListener(this);
  }

  protected void deactivate(ComponentContext context) {
    logReaderService.removeLogListener(this);
  }

  public void logged(LogEntry entry) {
    // Specific handling of the LogEntry
  }
}
```

Listing 119: Example implementation of a `LogListener` using the `LogReaderService`

For the same reason as for the Log service, we had to define a component description in a new folder *OSGI-INF* with the file *component-description.xml*. This file contains the component description, like shown in **Listing 120** below. Otherwise the use of the LogReader service is not possible.

```
<?xml version="1.0"?>
<component name="consoleLogListener">
  <implementation
      class="org.herasaf.tutorial.personlocator.logtracker.
             ConsoleLogListener" />
  <reference name="logService"
      interface="org.osgi.service.log.LogReaderService"
      bind="setLogReaderService"
      unbind="unsetLogReaderService" />
</component>
```

Listing 120: Component description of the LogReader service

The component description had to be added to the bundle manifest too, like described above.

6.7 Testing

Overview

In this chapter the test related topics of the developed reference architecture are described. This covers beside the solution of the unit and integration tests also the comparison of possible alternatives and the decision for one of them.

6.7.1 Unit Test

Overview

The unit tests verify the bundles functionality in a local scope, outside of the OSGi environment. Thus, to cover the bundle's functionality with tests, common

unit tests based on JUnit or TestNG can be used.

There surely are cases, where OSGi classes, such as a *BundleContext* or a *ServiceReference*, will be needed to complete a test case. To simplify this issue, Spring DM provides OSGi mocks under *org.springframework.osgi.mock*. However, by following best practices and using the Spring DM support, the classes should be easy to test as they won't have any hard dependencies on OSGi. If there are dependencies anyway, these are interface-based and therefore easy to mock.

Our solution

In our solution the unit tests are located in separate projects. These projects are common maven projects containing dependencies to test frameworks, like e.g. TestNG, to other packages required to accomplish the tests and to the bundle being tested.

The cause of having separate projects for the unit tests are the problems occurred in relation with the *Maven Classpath Container* and the *Plug-in Dependencies Container* in Eclipse. Having the unit tests in the same project as the OSGi bundle's project, can lead to duplicate classpath entries in the containers. This happens if e.g. a service client defines a dependency to a service. This dependency will appear in both containers causing a duplicate classpath entry. Another problem is that the maven dependencies are visible from the classes inside the OSGi bundle.

Unit Tests with TestNG and Spring

TestNG gives the possibility to create unit tests using a predefined Spring Context, which is used to auto-wire dependencies in the test case implementations. This allows keeping the test classes thin and being flexible enough to handle unpredictable changes easily.

To create a test in the tutorial project, an extension of the class `AbstractTestNGSpringContextTests` has been implemented. Using predefined annotations, the test suite, the tests and the path to the Spring application context are specified. For further details on TestNG and its functionality we'd like to refer to the project's website and documentation *[TestNGDoc]*.

Tests are defined in the *src/test/java* directory whereas Spring application contexts and other resources are placed under *src/test/resources*. To run the tests inside Eclipse we use the *TestNG Eclipse plug-in*. The plug-in allows JUnit like execution of tests. Further information about the installation and use can be found on the project's website *[TestNGDoc]*.

The automatic activation of the tests is controlled by Maven. Including the unit test project as a *module* in the *pom.xml* file of the parent project, assures the tests to be compiled and executed whenever the maven goal *test* is called. If problems related to TestNG occur, check if the proper TestNG packages and versions are available in the local Maven repository.

6.7.2 Integration Test

Overview

Whereas the unit tests verify only the isolated bundles where they belong to, the integration test verify the whole workflows particularly with regard to the dependencies between the bundles.

The special characteristic of these integration tests is the requirement for the availability of a target platform. This means to run the integration test, they use an OSGi environment. While other projects create mocks to hide or simulate the

frameworks they use, this is not or only possible with a huge effort in an OSGi environment.

Our solution

In the chosen solution the integration tests are located and implemented in own projects, which are independent from the ones who will be tested. Through the use of the Spring DM, the project doesn't have to be an OSGi bundle. Its testing framework creates an on-the-fly bundle for the test cases. Additionally, it starts the OSGi framework, adds the required bundles, run the tests and tears down the framework at the end. This allows a simplified test implementation compared with one where these steps have to be done manually.

Another big advantage of the Spring DM testing framework is the definition of the target platform. You don't have to define it, because it gets the required bundles for the target platform out of the local maven repository. The only thing that has to be done is to provide them in the local Maven repository.

Further reasons for this decision are the following:

- These days, it is best practice to separate the integration test from the tested code base, because it contains several classes and configurations which have no clear affiliation to a specific bundle.
- The integration test will run in a predefined OSGi environment, which assures real conditions for your tests. There is no need to mock environment instances in order to get a proper result.
- The Spring DM testing framework is very configurable concerning the creation of the on-the-fly test bundle and the execution of the OSGi environment. It allows the specification of an own manifest file, the use of a spring application context to wire the dependencies in the test classes, the addition of test bundles to be installed located from the local maven repository, referring to only a small set of possible configurations available.

Using Spring DM

To use the automations and simplifications of the Spring DM, the test cases have to override the `AbstractConfigurableBundleCreatorTests` class. It provides additionally access to variables like the `BundleContext` or allows the configuration of the on-the-fly bundle by overwriting methods. The test cases itself can be written like normal unit tests.

Additional Test Bundles

As mentioned above, the Spring DM takes the bundles for the target platform out of the local Maven repository. But this includes only the basic bundles. If additional ones are used, the method `getTestBundlesNames` has to be overridden. Its return value is a String array which contains the additional required bundles in the same description like Maven uses it. An example is shown below.

```
@Override
protected String[] getTestBundlesNames() {
  return new String[] {
    "org.herasaf.tutorial.calculator, control, 1.0.0",
    "org.herasaf.tutorial.calculator, core, 1.0.0",
    "org.herasaf.tutorial.calculator, services, 1.0.0",
    "org.herasaf.tutorial.calculator, service.impl, 1.0.0" };
}
```

Listing 121: Snippet of the `getTestBundlesNames` method out of an integration test

Specific MANIFEST.MF file

In our case we use the controller, the intermediation component out of the MVC Pattern, as the interface to the system under test. This means, that the test cases use the controller and/or the model to test the system for correct operations. Referring to these circumstances we have dependencies to different packages in the whole project, similar as in the UI bundle, to be able to test the system properly.

Due to problems during the execution of the tests, we found out that the manifest file, which is also generated on-the-fly at startup, doesn't contain all required bundles from the project to run correctly. The Spring DM testing framework provides hook methods to either override the generated (override `getManifest` method) or to specify the location of an own manifest file (override `getManifestLocatioin` method).

In our case we override the `getManifestLocation` method and provide an own MANIFEST.MF file in the classpath.

```
@Override
protected String getManifestLocation() {
    return "classpath:/MANIFEST.MF";
}
```

Listing 122: Snippet of the `getManifestLocation` method out of an integration test

This solution allows us to have the manifest file outside of the test code and therefore independent and to avoid complex program code to handle additional package entries.

Please note that when defining an own manifest file, it additionally needs to contain the following entries in order to work properly:

```
Bundle-Activator:
  org.springframework.osgi.test.JUnitTestActivator
Import-Package: junit.framework,
  org.osgi.framework,
  org.apache.commons.logging,
  org.springframework.util,
  org.springframework.osgi.service,
  org.springframework.osgi.util,
  org.springframework.osgi.test,
  org.springframework.context
```

Listing 123: Snippet of entries required in the manifest file for an integration test

Using a specific application context

Looking more precisely at the integration test, it's actually a kind of "simulation" of the UI. And in the UI we use Spring DM to wire the dependencies in the views, controllers, etc. This means that the implementation of the views, controllers and other classes uses attributes which are injected by Spring DM, meaning that there are no "new" operations used.

In order to be able to use the existing code in the controllers, models and other classes, we use Spring DM to wire the dependencies in the integration tests, too. To do this the method `getConfigLocations` has been overridden. The following code snippet shows how a specific application context has been defined.

```
@Override
protected String[] getConfigLocations() {
  return new String[] { "classpath:/testContext.xml" };
}
```

Listing 124: Snippet of the `createApplicationContext` method out of an integration test

The application context is available through a variable called *applicationContext* inside of the test class. By using this attribute, the beans specified in the application context can be accessed and used for the tests, as the following code snippet demonstrates.

```
@Override
protected void onSetUp() throws Exception {
  super.onSetUp();

  controller = (CalculatorController) applicationContext
    .getBean("calculatorController");

  model = (CalculatorModel) applicationContext
    .getBean("calculatorModel");
}
```

Listing 125: Snippet of the `onSetUp` method out of an integration test

Other configurations

By following the same principle, there are a lot of other configurations that can be applied for integration tests. For a detailed description, we'd like to refer to chapter 9 in the Spring DM reference guide [SpringDM]. As seen above, the configurations are defined by overriding methods in the frameworks super classes. Therefore it's always helpful to browse through the class hierarchy of the testing framework and to look for additional configuration hooks.

In the following subchapters the design decisions for the solution described above are listed in detail. They show the possible alternatives with its advantages and disadvantages as well as a detailed decision. It should provide you a better comprehension of our decision.

6.7.2.1 Possible solutions

Overview

To create integration tests which verify several or all parts of the system in a comprehensive manner, different solutions exist. Different than testing a normal java application, the integration tests in an OSGi environment are much more difficult, because the setup and tear down have to manage the OSGi framework.

The simplest solution to test the different parts of the application and its dependencies among each other is to implement simple unit tests and to simulate the OSGi framework where necessary. Other solutions are text cases in a running OSGi environment with all its advantages and disadvantages.

Now we take a look at some possible solutions. Besides a general description of the solutions we also compare the advantages and disadvantages in detail.

Alternatives

Alternative 1: Integration tests without the OSGi framework

The first solution to create integration tests is to run them without the OSGi framework. In this case it's necessary to simulate the OSGi framework, because some of the used functions and methods are provided by classes of the OSGi framework. Since the OSGi framework makes extended use of interfaces the required classes can be mocked and injected with the Spring dependency injection.

Advantage	Disadvantage
• The OSGi framework is not needed which allows decreases the configuration effort and the complexity of the integration tests. • As no OSGi framework is used, we don't need a target platform to run the integration tests. Only the needed bundles and libraries have to be available. • The embedding of the integration tests in an automated build environment can be achieved very easily.	• Additional mock implementations are needed to simulate the OSGi framework. This can lead to an increased configuration effort. This can neutralize the configuration savings. • Only in an OSGi environment the correct visibility of the packages can be tested, because only OSGi interprets the extended manifest information. Otherwise all packages can be accessed without the verification of this restriction.

Alternative 2: Integration tests as bundle using a testing framework

A possible solution for integration tests is to implement them in a bundle and to run it in an OSGi environment. This requires from the integration tests to set the OSGi framework up to run the test cases and to tear it down afterwards. The configurations and implementations to do that are substantial. Additionally they are not tests in the true sense of the integration testing, but simple infrastructure implementations.

To use the OSGi environment a target platform with the required bundles is needed. Before the tests can be executed, the target platform has to be available. Otherwise the OSGi framework cannot be set up and started.

Advantage	Disadvantage
• The full support of the OSGi framework can be used, like the service registry and further more. • The test cases run in a real-life environment. This allows testing close to reality. • No special mock classes are needed to simulate the functionality of the OSGi framework.	• To run the test cases, they have to be implemented as bundles. Otherwise they cannot be executed in the OSGi framework. • The definition of a target platform is required, where the bundles and the integration tests can be installed and executed on. Before executing them, the target platform has to be available on the system which implies a mechanism to provide it. This leads to configurations and implementations which are substan-

	tial.
	• An automation of the integration tests with Maven or other build tools will be very hard to provide, cause of the required target platform as well as the set up and tear down of the OSGi framework. • The embedding of the integration tests in an automated build environment can be hard to achieve. It's up to your solution if it's possible or not.

Alternative 3: Integration tests with Spring DM support

The third applicable solution to create integration tests for applications in an OSGi environment is the use of the Spring DM test support. Generally, it's the solution above where you implement the integration tests as own bundles but with the support of the testing framework provided by the Spring DM, which means a great simplification. The integration tests don't have to be written as bundles, because Spring DM creates an on-the-fly bundle when the integration test is executed. Additionally, the target platform setup is simplified to a minimum, because you only have to provide the bundles of it in the local maven repository.

The other steps for setting up and starting the platform before executing the integration tests are handled by Spring DM as well as the tear down and clean up afterwards.

Advantage	Disadvantage
• The full support of the OSGi framework can be used, like the service registry and further more. • The test cases run in a real-life environment. This allows testing close to reality. • No special mock classes are needed to simulate the functionality of the OSGi framework. • The configuration of the target platform can be minimized, because the Spring DM uses the bundles in the local maven repository. They only have to be downloaded in these directories. • The embedding of the integration tests in an automated build environment like Maven	• The Spring DM testing framework requires a local maven repository. This forces the use of Maven. • The target platform has to be configured in the pom.xml. Otherwise the required bundles aren't available in the local maven repository.

can be achieved very easily,
because it cooperates with it.

Decision

Based on the comparisons shown above, we decided to implement our integration tests as bundles with the Spring DM support. The reasons for this decision are the simplicity of their configuration and implementation as well as the comprehensive use of all OSGi features and functions. The manual implementation of the whole setup and tear down procedures, which causes substantial efforts, are not central parts of the integration test creation.

That's the reason why the second solution which also uses the OSGi framework to execute the integration tests, but requires a manual configuration and implementation of the whole infrastructure setup and tear down is not appropriate.

Also the first solution which propose to simulate the OSGi framework where necessary and outside of an OSGi environment is not appropriate. It also causes too much extra efforts, which are not necessary and can bring imponderable risks with it. That's why this solution is also not appropriate or only in exceptional cases.

6.7.2.2 Problems

Additional Test Bundles

The configuration of the required test bundles for an integration test in the test class itself is complicated and error-prone. Especially in cases where an automatic build environment is used and the required bundles and plug-ins are configured anyhow is a duplicate configuration of them nonsense.

Therefore we tried to implement an automatic configuration of the integration tests. The *pom.xml* file contains already all the required bundles and plug-ins, so it's obvious to use its dependencies. Since we outsourced the used version numbers in the parent project, a simple load of the file through the test class is not possible. But the *Maven Dependency Plugin* allows the export of the dependency list in a file which can be parsed.

The parsing of the generated dependencies in the test bundle is also possible but depends strongly on the used Maven plug-in. Additional problems occurred during the duplicate configuration of bundles, because some of the bundles are contained in the basic configuration of the Spring DM testing framework. So the filtering of them is necessary.

All in all, the automatic configuration of the integration tests with the dependencies out of the *pom.xml* file would lead to a fragile solution. Not only the strong dependence to the used Maven plug-in, and especially the used version, is error-prone, also the configuration of the bundles and plug-ins, which should be filtered, eliminates the possible advantages.

That's why we decided to implement the integration test without an automatic configuration.

Several test classes

Another very cumbersome problem we had during the implementation of integration tests were inexplicable exceptions. After some trial we found the cause of the problem. It seems the Spring DM testing framework is not able to handle the execution of more than one test class if they have different configurations. This means different test bundles, different application contexts and a different manifest. But a description of such a restriction is not contained in the Spring

Dynamic Modules documentation *[SpringDM]*.

To solve the problem, we decided to create an abstract super class defining the manifest, application context and bundles configurations, like shown in **Listing 126** below. For the implementation of the specific test cases we inherited from the abstract class. This was the only way we could locate several test classes in a single project. Unfortunately these configurations cannot be separated anymore for each test class.

```
public abstract class AbstractServiceTest extends
    AbstractConfigurableBundleCreatorTests {

  @Override
  protected String[] getConfigLocations() {
    return new String[] {
       "classpath:/bundleContext-dictionary.xml",
       "classpath:/bundleContext-domain.xml"};
  }

  @Override
  protected String getManifestLocation() {
    return "classpath:/MANIFEST.MF"; //$NON-NLS-1$
  }

  @Override
  protected String[] getTestBundlesNames() {
    return new String[] {
        "org.herasaf.tutorial.personlocator, core, 1.0.0",
        "org.herasaf.tutorial.personlocator, service,
            1.0.0",
        "org.herasaf.tutorial.personlocator,
            service.domain, 1.0.0",
        "org.herasaf.tutorial.personlocator,
            service.dictionary, 1.0.0" };
  }
}
```

Listing 126: Example of an abstract super class for the integration test implementation

6.8 Configuration

Overview

In this chapter the configuration related topics of the developed reference architecture are described. This covers beside the concrete solution also the comparison of possible alternatives and the decision for one of them.

6.8.1 Bundle dependency types

Overview

In the development of applications it's obvious to use classes and interfaces out of other libraries. In an OSGi environment a bundle can use not only libraries but also other bundles and plug-ins. To use the classes out of them, they must be imported in such a way through a directive in the bundle manifest file. Otherwise the classes and interfaces are not accessible.

Therefore the OSGi compliant manifest provides different kinds of import directives. The use of them and especially the decision which one has direct impact

on the adaptability and stability of your bundle or plug-in.

Import-Package

The most recommended and therefore the most used manifest directive is the *Import-Package*. Its big advantage is the dependency to the other bundles, because it's totally different from everything else. Instead to import the whole bundle, only a selection of its packages is imported. Additionally, these packages must be exported from the bundle which contains them. This allows a fine-grained management of the public and private classes instead of importing everything from the bundle. Thus it reduces the strength of the dependencies between the bundles.

As an additional advantage the OSGi environment takes the responsibility to provide your bundle with the required classes and interfaces. It scans all the installed bundles and chooses the right ones to satisfy the import directive. In principle you don't have to know which bundle exactly contains these classes.

For this reasons we also used the *Import-Package* directive in most cases. It allowed us to provide our bundle with the required classes but without the need to have a strong dependency to a specific bundle.

But unfortunately we had some troubles with the *Import-Package* directive because of the split packages. These are packages containing classes but split into different bundles. Normally that's no problem but in combination with the *Bundle Plug-in for Maven*, the manifest was generated with the *…=split* option. This option forces OSGi to import the classes from only the bundle declared before the equals. This can be avoided through the import of the bundles with the *Require-Bundle* directive.

DynamicImport-Package

A special kind of the *Import-Package* directive is the *DynamicImport-Package*. It has the same advantages and disadvantages like the *Import-Package*. Its additional intent is to look for an exported package only when that package is really needed. The OSGi Core specification *[OSGiCore]* describes its main use like this:

"The key use case for dynamic import is the Class forName method when a bundle does not know in advance the class name it may be requested to load."

We used this directive hardly, because we worked very seldom with reflection in our tutorials. But it is a nice feature to define dependencies on-demand, which means only in cases they are really used.

Require-Bundle

The complete opposite of this relative lose dependency is the *Require-Bundle* directive. It is the strongest dependency between bundles in an OSGi environment, because it results in a complete import of all classes and interfaces contained in the referenced bundle. Therefore it has the disadvantage of a dependency to a specific bundle.

Therefore, we decided to avoid its use as much as possible, but in cases where extension points were required for the implementation in RCP applications or split package problems occurred, we used this directive.

7 Processes

Overview

Concerning the use of our reference architecture we also made research on the required development processes. Well designed and thought-out processes help the developer to be effective and efficient in his tasks.

In this chapter we describe the processes for the use of the reference architecture developed in this thesis.

Intention

A process describes the required steps to achieve a specific goal. In most cases it covers all the existing application layers and gives the developer instructions for an effective and efficient use of the reference architecture developed in this thesis. Thus, the intention of this chapter is to show what these processes consist of and how they can be used in practice.

7.1 Setup of the development environment

Objectives

The development environment consists of the tools which allow creating a concrete product with the specific technologies. Setting up these tools in order to work together and to behave as expected can be a complicated and time-consuming task.

The goal of this process is to create a base for the development of applications and to provide a flexible and easy to use programming environment.

Process

The process consists of the following steps:

1. Download and install Eclipse for RCP/Plug-in Developers
2. Download the base RCP bundle package and compose a custom target platform based on your requirements
3. Include/add extra bundles required in your target platform
4. Configure Eclipse to use the custom target platform
5. Introduce and configure the automatic build environment

Details about these steps are documented in the following parts of this chapter.

Eclipse Installation

The Eclipse IDE for RCP and Plug-in developers can be downloaded on the Eclipse Project website. The installation is straightforward and detailed instructions can be found on the Eclipse Project website, too.

Target Platform

Also on the Eclipse Project website a base RCP package can be downloaded. It includes the basic Eclipse/RCP bundles and plug-ins which are required to develop a standalone rich client application.

Depending on your needs and requirements, further bundles and plug-ins can be added to the base RCP package. The Spring DM bundles are a good example of additional bundles.

Having your target platform composed, the next step is to save the target platform in an own Eclipse project. This is done by creating a simple project in Eclipse and copying the target platform, including all files and folders, into it. A target definition description file is required to describe your target platform and to allow an easy Eclipse configuration. The target definition file can be chosen in the target platform configuration tab in the preferences dialog of Eclipse. By loading the target definition file, Eclipse is directed to use your custom target platform for the plug-in development.

Because the target platform is the base for the further plug-in development, we recommend uploading it into the SVN repository. Like this, a developer simply can check it out, configure its Eclipse and has the appropriate base for the further implementation.

Automatic build

The automatic build is described in chapter 7.1.1.

7.1.1 Setup the automatic build environment

Objectives

An automatic build is intended to make the development easier by completing structured and repetitive tasks automatically. The objectives of it are the integration of all available projects in the build process, the execution of tests and the automatic creation of ready-to-use products automatically at any time.

Preconditions

In order to be able to setup an automatic build environment, a fully functional development environment has to exist.

Process

The process consists of the following steps:

1. Download, install and configure Maven
2. Integration of the Apache Felix *Bundle Plug-in for Maven* in Maven
3. Configuration of the Headless PDE Build and its integration in Maven
4. Inclusion of integration and unit tests into the build process

Details about these steps are documented in the following parts of this chapter.

Maven

Detailed instructions about the installation and configuration of Maven can be found on the Apache Maven website *[Maven]* as well as in chapter 8.1.3.

Maven Bundle Plug-in

The *Bundle Plug-in for Maven* generates the manifest files for bundles during the build process automatically based on the configurations in the *pom.xml*. Like this, it makes the integration in the Maven build process possible and gives further advantages. A detailed description of it can be found in chapter 8.1.5.1.

Headless PDE Build

In the case of RCP applications the PDE build generates a ready-to-use application including all the required bundles and plug-ins as well as an executable file. It is an important feature which completes the automatic build process and makes it possible to create the whole application automatically. A detailed de-

scription about this topic can be found in chapter 8.1.5.2.

Testing

The inclusion of integration- and unit tests in the build process controls changes during the development and makes sure these changes are secure and functional. Additionally it prevents the developer from uploading instable code to the central repository and thus raises the code quality. A detailed description about testing can be found in chapter 8.7.

7.2 Create a RCP application

Objectives

The objective of this process is a standalone RCP application which is implemented based on the research and the best practices developed in this thesis. This application is the base for the further development and therefore this process includes some extra steps for the creation of core and service bundles.

Preconditions

In order to be able to create a RCP application, the development environment has to be setup correctly and ready-to-use.

Process

The process consists of the following steps:

1. Create a standalone RCP application project in Eclipse
2. Convert the project to a Maven project
3. Configure and define the plug-in extensions (*plugin.xml*)
4. Create and configure the product description file (.product file)
5. Modify main application classes to adapt to your requirements
6. Adjust the build properties and startup configuration
7. Create a Maven root project
8. Create a core project
9. Create a language project
10. Create a service layer

Details about these steps are documented in the following parts of this chapter.

RCP project

The Plug-in Developers edition of Eclipse provides special plug-in projects for the creation of standalone RCP applications. These projects are supported by the PDE with different tools, like e.g. plug-in configuration, execution, creation of products and further more.

To create a standalone RCP application, you have to add a new *plug-in* project. In the creation wizard, the question “Would you like to create a rich client application?” should be set to *yes*.

Convert to Maven project

Converting the project to a Maven project allows the integration of it in the automatic build process. After the conversion, the *Maven Classpath Container*

should be removed to avoid duplicate entries in the project classpath.

The configuration of Maven and the *Bundle Plug-in for Maven* ensures that the required dependencies are set and enables the automatic build to run correctly.

Plug-in extension

After the creation of the standalone application, its extensions should be configured. This is through Eclipse's extension point mechanism by defining extensions in the *plugin.xml* plug-in description file. Like this, various functionalities, like e.g. the menus and toolbars, can be configured.

Which extension points to use depends on the requirements of your application. The use of the extension points is documented in various chapters in the description of the solution (see chapter 6.3.1) and the developer's guide (see chapter 8.3.1).

Product description

The next step is the definition of a product description. This is done by adding a *.product* file into your RCP standalone application project. The file is mainly used for the execution of the product and for the PDE build.

A more detailed description about this topic can be found in *[Daum2008]*.

Main application classes

At the creation of a standalone Rich Client Application, Eclipse generates the default application classes automatically. These classes can be modified to customize the application in order to fulfill your requirements.

In chapter 8.3.2 a detailed description about this topic can be found.

Build and startup configuration

The RCP standalone application project is the base project concerning the user interface and the execution of the application. Therefore, this project has to assure the correct startup of the OSGi environment including all the required bundles.

In this step, the *config.ini* and the *build.properties* files should be checked and modified in order to meet you requirements. A more detailed description about this topic can be found in chapter 8.1.5.2.4.

Maven root project

Having the RCP application bundle configured completely, the next step is to create a Maven root project. This project specifies all the modules of the whole application and acts as a starting point for the automatic build.

The Maven root project is a Maven project in Eclipse and should be configured using the *pom.xml* descriptor. A detailed description about this topic can be found in chapter 8.1.5.1.

Core project

The next step is to create a Maven project representing the core project which contains the domain objects of the application and other central parts.

Because these objects are used in multiple bundles from different application layers, it is appropriate to create an own bundle for them.

A detailed description about this topic can be found in chapter 8.1.5.1.

Language project

A user interface uses text descriptions for the various widgets and functions it provides. Thus, the next step is to externalize such description strings into a language bundle in order to make language changes easy and to centralize its configuration.

A detailed description about this topic can be found in chapter 8.3.13.

Service layer

The developed reference architecture in this thesis specifies the use of a service layer. In this step a service root project should be created which includes the service interfaces for the application.

As for the implementation of the services, for each aspect an own project should be created which includes the implementation classes.

More about this topic is available in chapter 8.5.

7.2.1 Wiring a RCP application

Objectives

The goal of this process is to introduce Spring DM in your project in order to raise the flexibility and reusability.

Process

The process consists of the following steps:

1. Install the Spring libraries in the OSGi environment
2. Configure the Spring application context
3. Inject the beans using the Spring extension factory

Details about these steps are documented in the following parts of this chapter.

Spring libraries

In order to be able to use the Spring techniques, the specific libraries have to be downloaded, included in the target platform and activated when the OSGi environment starts up.

Details about this topic can be found in chapter 8.1.2.2.

Spring application context

The configuration of the application context should be specified in *.xml* files contained in the *META-INF/spring* folder. By default, the Spring DM bundles search this folder for application context descriptions.

A more detailed description about this topic can be found in chapter 8.3.14.

Spring extension factory

As already mentioned before, the extension point mechanism of Eclipse is used to make extensions concerning the user interface. In order to be able to inject Spring Beans for the extension of extension points, the *Spring Extension Factory* should be used.

A more detailed description about this topic can be found in chapter 8.3.14.

7.3 Add a new plug-in to an existing RCP application

Objectives

After the execution of this process, a new RCP plug-in is added to an existing RCP application. Plug-ins are used to extend the application modularly.

Preconditions An RCP standalone application has to exist.

Process The process consists of the following steps:

1. Create a Maven project in Eclipse
2. Convert the project to a Plug-in project
3. Configure the bundle
4. Remove the Maven Repository in the project
5. Integration in the remaining application

Details about these steps are documented in the following parts of this chapter.

Maven project The first step of this process is to create a Maven project in Eclipse. By doing this, a *pom.xml* descriptor is available and allows the integration of it in the automatic build process.

Convert to plug-in project By converting the project to a plug-in project, RCP specific configurations are provided, which are required to integrate this additional plug-in into the whole RCP application.

Bundle configuration The next step is to make configurations concerning Maven, the *Bundle Plug-in for Maven* and the plug-in itself. These configurations are required for the correct integration of the plug-in in the whole RCP application.

An example about the bundle configuration can be found in the chapters 16 and 17.

Maven classpath The *Maven Classpath Container* has to be removed to avoid duplicate entries in the project classpath. A detailed description about this topic can be found in chapter 8.1.5.1.3.

Integration As the last step, the new project has to be added in the remaining application. For this purpose, it must be added as a reference in the *pom.xml* of the root project.

7.4 Create a TextEditor

Objectives After the execution of this process a text editor is added to your application and can be used for predefined file extensions.

Preconditions In order to be able to create a text editor, a RCP application has to be available.

Process The process consists of the following steps:

1. Create a new plug-in for your RCP application
2. Add and configure extensions
3. Implement the required classes for the extensions
4. Use Spring DM to wire the implemented classes
5. Create control layer

Details about these steps are documented in the following parts of this chapter.

Plug-in project

The addition of a further plug-in to an existing RCP application is described in chapter 7.3.

Plug-in extensions

The affected extension points for the extension of the RCP application with a text editor are the following:

- org.eclipse.ui.editors
- org.eclipse.ui.commands
- org.eclipse.ui.handlers
- org.eclipse.ui.menus

These extension points are required in order to be able to make advanced adjustments. A detailed description about this topic can be found in chapter 8.3.9.

Extension classes

Most of the extension points require the specification of concrete classes implementing predefined interfaces. These classes can be own implementations as well as default implementations provided in the framework.

The next step is to create these classes in order to customize the behavior of the text editor concerning your requirements. A more detailed description about this topic can be found in chapter 8.3.9.

Spring DM

The wiring of a RCP application is described in chapter 8.3.14.

Control layer

The use of a control layer decouples the view implementation from the implementation of its behavior. To take this advantage a new Maven project should be created in Eclipse and configured as shown in chapter 7.3.

This bundle contains a controller and takes the responsibility to communicate with the service layer and use a parser for the conversion of the data. A more detailed description of this topic can be found in chapter 8.4.

7.4.1 Extend a TextEditor with additional features

Objectives

This process describes how additional features can be added to an existing text editor in your application. The objective of this process is to have a text editor which supports the user with different features and is configured based on your requirements.

Preconditions A text editor has to be available in an existing RCP application in order to be able to extend it with further features.

Features The text editor in the Eclipse environment can be extended amongst others with the following features:

1. Add syntax highlighting
2. Introduce a model reconciliation strategy
3. Provide an outline page
4. Introduce content assistance
5. Add annotations
6. Work with the context menu and items

Detailed description A detailed description about these features can be found in chapter 8.3.9.

7.4.2 Change of the domain model behind the TextEditor

Objectives This process describes how the domain model behind the text editor can be changed and what steps need to be taken in order to achieve this goal. The objective of this process is to have the text editor working with the new domain model at the end.

Process The process consists of the following steps:

1. Change the domain model in the core project
2. Adapt the service implementations to the new circumstances
3. Adapt the control layer to the new circumstances
4. Write integration and unit tests to cover the changes

Details about these steps are documented in the following parts of this chapter.

Change domain model The first step of this process is to change the model in the core project to fulfill your new requirements. Because the core project is used by several bundles in the application it implies changes in the appropriate projects.

Services The services working with the changed domain objects should be modified in this step. The services get used in the whole application, thus it is an important task to adapt them to the new circumstances and assure the work properly.

Controllers Controllers carry the responsibility of the communication with this service layer and the user interface layer. Thus, the next step is to adapt the controller beha-

vior, especially the parsers, to work with the changes in the domain model correctly.

Testing

It is obvious that to ensure the proper function of the service and controller layer, the creation of integration and unit tests is the way to go.

7.5 Create a graphical editor

Objectives

This process describes how a graphical editor can be added to an existing RCP application. The objective of this process is to have a graphical editor which supports the user to accomplish his tasks and is configured based on your requirements.

Preconditions

In order to be able to create a graphical editor, an RCP application has to be available.

Process

The process consists of the following steps:

1. Create a new plug-in for your RCP application
2. Add and configure extensions
3. Implement the required classes for the extensions
4. Select and implement further features
5. Use Spring DM to wire the implemented classes
6. Create control layer

Details about these steps are documented in the following parts of this chapter.

Plug-in project

The addition of a further plug-in to an existing RCP application is described in chapter 7.3.

Plug-in extensions

The affected extension points for the extension of the RCP application with a text editor are the following:

- org.eclipse.ui.editors
- org.eclipse.ui.commands
- org.eclipse.ui.handlers
- org.eclipse.ui.menus

These extension points need to be used in order to be able to make advanced adjustments. A detailed description about this topic can be found in chapter 8.3.9.

Extension classes

Most of the extension points require the specification of concrete classes implementing predefined interfaces. These classes can be own implementations

as well as default implementations provided in the framework.

The next step is to create these classes in order to customize the behavior of the text editor concerning your requirements. A more detailed description about this topic can be found in chapter 8.3.9.

Further features

In this step further features should be implemented for the graphical editor depending on your requirements. A more detailed description of this process can be found in chapter 7.5.1.

Spring DM

The wiring of a RCP application is described in chapter 8.3.14.

Control layer

The use of a control layer decouples the view implementation from the implementation of its behavior. To take this advantage a new Maven project should be created in Eclipse and configured as shown in chapter 7.3.

This bundle contains a controller and takes the responsibility to communicate with the service layer and use a parser for the conversion of the data. A more detailed description of this topic can be found in chapter 8.4.

7.5.1 Extend the graphical editor with further features

Objectives

This process describes how additional features can be added to an existing graphical editor in your application. The objective is to have a graphical editor which supports the user with different features and is configured based on your requirements.

Preconditions

A graphical editor has to be available in an existing RCP application in order to be able to extend it with further features.

Process

The text editor in the Eclipse environment can be extended amongst others with the following features:

1. Add a palette providing tools for the editing and creation of diagrams
2. Introduce an outline view
3. Create and configure the context menu and action bars
4. Provide zoom functionality
5. Define key handlers
6. Provide direct edit functionality
7. Introduce drag & drop
8. Create a miniature view

Detailed description

A detailed description about these features can be found in chapter 8.3.8.

7.5.2 Extend the domain model of the graphical editor

Objectives

In cases where the core domain model changes and/or additional functionality in the graphical editor are desired, its editor model has to be changed.

This process describes how the specific domain model of the graphical editor should be extended. The objective is to have a fully adapted graphical editor at the end which can handle the new circumstances.

Preconditions

A graphical editor has to be available in an existing RCP application in order to be able to extend its domain model.

Process

The process consists of the following steps:

1. Change the domain model in the graphical editor
2. Adapt or create dependent components in the graphical editor
3. Modify the existing features of the graphical editor
4. Adapt the control layer to the new circumstances

Details about these steps are documented in the following parts of this chapter.

Graphical editor model

In this step the domain model of the graphical editor has to be changed to fulfill the new requirements. A more detailed description of this topic can be found in chapter 8.3.8.4.2.

Graphical editor components

After the change of the model in the graphical editor structure with the other MVC components have to be adapted to the new circumstances. It is required to check and/or modify the edit parts as well as the figures in your implementation.

A more detailed description of this topic can be found in chapter 8.3.8.4

Graphical editor features

Having adapted the graphical editor's MVC structure and its parts, the editor features have to be checked, too. Depending on the weight of the changes it can be necessary to add new features to the existing graphical editor.

A more detailed description about this process is available in chapter 7.5.1.

Control layer

Depending on the dimension of the changes it can be necessary to adapt the control layer in order to match the extensions in the graphical editor model. Especially the parsers, which are used to convert from the core domain model to the graphical model and vice versa, require modifications in such a case.

In the chapters 8.4 and 8.3.8.3 a more detailed descriptions about this topic are available.

7.5.3 Use an external domain model

Objectives

This process describes how an external model, like e.g. the core domain model,

can be integrated in the graphical editor. The objective is to have an editor at the end, which uses a specific external domain model and is able to show and make manipulations on it.

Preconditions

A graphical editor has to be available in an existing RCP application in order to be able to integrate an external domain model into it.

Process

The process consists of the following steps:

1. Define the graphical model for the visualization
2. Adapt or create dependent components in the graphical editor
3. Modify the existing features of the graphical editor
4. Adapt the control layer to the new circumstances

Details about these steps are documented in the following parts of this chapter.

Graphical model

In a first step, the graphical model for the visualization of the different parts has to be defined. Because it highly depends on the requirements of your application, it should be specified based on your needs.

Such a specification of a graphical model implies the extension of the existing graphical editor domain model which is described in chapter 7.5.2.

Graphical editor components

After the adaption of the existing model to the external model, the other MVC components have to be adapted to the new circumstances. It is required to check and/or modify the edit parts as well as the figures in your implementation.

A more detailed description of this topic can be found in chapter 8.3.8.4

Graphical editor features

Having adapted the graphical editor's MVC structure and its parts, the editor features have to be checked, too. Depending on the weight of the changes it can be necessary to add new features to the existing graphical editor.

A more detailed description about this process is available in chapter 7.5.1.

Control layer

Depending on the dimension of the changes it can be necessary to adapt the control layer in order to match the extensions in the graphical editor model. Especially the parsers, which are used to convert from the core domain model to the graphical model and vice versa, require modifications in such a case.

In the chapters 8.4 and 8.3.8.3 a more detailed descriptions about this topic are available.

7.6 Create an OSGi Service

Objectives

This process describes how an OSGi service can be integrated in your application and which steps need to be taken in order to achieve this goal. The objective of this process is to have an OSGi service at the end which is registered in

the OSGi service registry and can be used application wide.

Process

The process consists of the following steps:

1. Create a Maven project in Eclipse
2. Convert the project to a Plug-in project
3. Configure the Plug-in
4. Remove the Maven repository in the project
5. Implement the service
6. Wire the service using Spring DM
7. Introduce a notification mechanism for changes

Details about these steps are documented in the following parts of this chapter.

Maven project

The first step of this process is to create a Maven project in Eclipse. By doing this, a *pom.xml* descriptor is available and allows the integration of it in the automatic build process.

Convert to plug-in project

By converting the project to a plug-in project, RCP specific configurations are provided, which are required to integrate this additional plug-in into the whole RCP application.

Plug-in configuration

The next step is to make configurations concerning Maven, the *Bundle Plug-in for Maven* and the plug-in itself. These configurations are required for the correct integration of the plug-in in the whole RCP application.

An example about the bundle configuration can be found in the chapters 16 and 17.

Maven classpath

The *Maven Classpath Container* has to be removed to avoid duplicate entries in the project classpath. A detailed description about this topic can be found in chapter 8.1.5.1.3.

Service implementation

The next step is the service implementation. First, the service root project has to be referenced in order to gain access to the service interfaces. Then, the new service should be implemented by using an existing interface or by creating a new one depending on your requirements.

Spring DM

The wiring of an OSGi service is required. Its description can be found in chapter 8.5.2.

Notification

The introduction of a notification mechanism is also required. It is described in chapter 7.6.2.

7.6.1 Wiring an OSGi Service

Objectives

The objective of this chapter is to introduce Spring DM in your service layer. Spring DM simplifies the implementation of OSGi services massively and increases the flexibility and reusability of the components.

Preconditions

A service layer including a service root bundle and an appropriate service implementation bundle should exist in order to be able to introduce Spring DM in your application layer.

Process

The process consists of the following steps:

1. Install the Spring libraries in the OSGi environment
2. Configure the Spring application context

Details about these steps are documented in the following parts of this chapter.

Spring libraries

In order to be able to use the Spring techniques, the specific libraries have to be downloaded, included in the target platform and activated when the OSGi environment starts up.

Details about this topic can be found in chapter 8.1.2.2.

Spring application context

The configuration of the application context should be specified in *.xml* files contained in the *META-INF/spring* folder. By default, the Spring DM bundles search this folder for application context descriptions.

A more detailed description about this topic can be found in chapter 8.3.14.

7.6.2 Introduce a Notification Mechanism

Objectives

After the execution of this process a notification mechanism is integrated in your application providing a synchronization of the controller and the services in the case of domain model changes.

Preconditions

A control layer, a service root bundle and appropriate service implementations have to exist in order to be able to introduce a notification mechanism between them.

Process

The process consists of the following steps:

1. Define the listener interfaces in the service root project
2. Configure the service provider
3. Configure the service client

Details about these steps are documented in the following parts of this chapter.

Listener interfaces

The first step is defining appropriate listener interfaces for the notification. Having a look of how the Whiteboard pattern works, these interfaces are used for the registration of listener in the OSGi service registry. See chapter 8.5.2 for more details.

Service provider

The service provider has to be configured to look up the OSGi service registry for listener services and to notify all of them in the case of domain model changes.

A service implementation bundle represents such a service provider. Its Spring application context is configured to refer to a list of listeners services and to publish a concrete service implementation for a specific service interface.

Service client

The service client, which in this case is mostly a controller in the control layer, implements the appropriate listener interface and registers this implementation as a listener service in the OSGi service registry. Like this, the controller is able to react on domain model changes, which are announced by the service implementations.

7.7 Create additional Integration Tests

Objectives

The objective of this process is to add further integration tests for the application. After the execution of this process additional integration tests are available and integrated in the automatic build process.

Process

The process consists of the following steps:

1. Create Maven project in Eclipse
2. Configure Maven
3. Implement the test classes
4. Configure the test classes
5. Create configuration files

Details about these steps are documented in the following parts of this chapter.

Maven project

The first step of this process is to create a Maven project in Eclipse. By doing this, a *pom.xml* descriptor is available and allows the integration of it in the automatic build process.

Maven configuration

Maven has to be configured in order the integration test to work properly. Additional dependencies are required to be able to use the Spring DM testing framework. It provides OSGi environment internal execution of tests and is configurable for specific needs.

An example about the configuration of Maven can be found in the chapters 16

and 17.

Test class

The Spring DM testing framework provides an abstract class called `AbstractConfigurableBundleCreatorTests` which provides all the required functionality in order to create a full integration test. It should be implemented and configured as described in chapter 8.7.2.1.

Test class configuration

As already mentioned before, a detailed description about how an integration test class should be implemented and configured, can be found in chapter 8.7.2.2.

Configuration files

For the execution of the integration tests inside of the OSGi environment, the Spring DM testing framework generates a temporary bundle. This bundle has to be configured like all other bundles, too. Therefore, a manifest definition as well as an appropriate Spring application context has to be specified.

A more detailed description of this topic can be found in chapter 8.7.2.2.

Test execution

The integration tests can be executed as simple JUnit tests making the integration in Maven unproblematic.

7.8 Create additional Unit Tests

Objectives

The objective of this process is to add further unit tests for the application. After the execution, additional unit tests are available and integrated in the automatic build process.

Process

The process consists of the following steps:

1. Create Maven project in Eclipse
2. Configure Maven
3. Implement the test classes
4. Configure the test classes
5. Create configuration files

Details about these steps are documented in the following parts of this chapter.

Maven project

The first step of this process is to create a Maven project in Eclipse. By doing this, a *pom.xml* descriptor is available and allows the integration of it in the automatic build process.

Maven configuration

Maven has to be configured in order the unit tests to work properly. Additional dependencies are required to be able to use the TestNG testing framework. It allows the creation of Spring context configured tests.

An example about the configuration of Maven can be found in the chapters 16

and 17.

Test class

The TestNG testing framework provides an abstract class called `AbstractTestNGSpringContextTests` which provides all the required functionality in order to create a complete unit test. It should be implemented and configured as described in chapter 8.7.1.

Test class configuration

The configuration of the test class can be done through the use of annotations in the TestNG testing framework. The application context location has to be defined in order to enable the framework to create the Spring configured application context at startup.

A more detailed description about this topic can be found in chapter 8.7.1.

Configuration files

The only configuration file in the case of TestNG based unit tests is the Spring application context descriptor. This file should be configured based on your requirements and specified in the test class.

A more detailed description about this topic can be found in chapter 8.7.1.1.

Test execution

The unit tests can be executed as TestNG tests, which are supported by Maven and thus are integrated in the automatic build process.

8 Developer's Guide

Overview This chapter contains instructions for the developers how to implement custom applications based on the best practices and recommendations made during the research and construction phase.

8.1 Infrastructure

Intention This chapter contains instructions and recommendations for the definition and setup of an own infrastructure as the base for the development of the reference architecture.

8.1.1 Java

Overview Base of the whole infrastructure is the Java Development Kit, or JDK for short. It is used for both, the Eclipse RCP/Plug-in development and for the execution of the developed bundles in the OSGi framework.

Setup To install the Java *JDK* follow the instructions below:

1. Download the *JDK 5.0* or above.
2. Install it and follow the instructions of the setup wizard.
3. Open the system properties and chose the tab *Advanced*.
4. Open the *Environment Variables*.
5. Add a new entry JAVA_HOME to the directory where you installed the JDK in step 2.

Used versions For this bachelor thesis the *JDK 1.5 Update 17* was installed and used.

8.1.2 Eclipse

Overview For the development of Eclipse Plug-ins with RCP and OSGi bundles, you need to install the *Eclipse for RCP/Plug-in Developers*. It allows an easy development and execution of bundles and plug-ins in the well-known Eclipse manner.

Setup To install the *Eclipse for RCP/Plug-in Developers* follow the instructions below:

1. Download the *Eclipse for RCP/Plug-in Developers.*
2. Extract the downloaded zip file into a directory of your choice.
3. Start Eclipse with a double click on eclipse.exe.

Used versions For this bachelor thesis the Eclipse 3.4.2 (Ganymede) was used.

8.1.2.1 Plugins

Overview

For an optimal development, your Eclipse has to be extended with several plug-ins. All of them can be installed using the Eclipse Software Update mechanism.

The following list shows which plug-ins you should install and what's the purpose of them:

- Spring IDE *[SpringIDE]*: The Spring IDE provides additional tools and views for the implementation of Spring AOP, application context files and furthers more.
- Maven for Eclipse *[IAM]*: This plug-in is required for the development with Maven. It includes the Maven configured libraries and bundles into the Bundle Classpath.
- SVN for Eclipse *[Subversive]*: The Subversive plug-in allows the SVN integration into Eclipse. You can add new resources to SVN, check-in and check-out resources seamless out of your Eclipse environment.
- TestNG *[TestNG]*: TestNG is an improved testing framework giving you a broader range of testing options, like the definition of an input method for a specific test method and so on.
- WickedShell *[WickedShell]*: The WickedShell extends the Eclipse instance with a command prompt like well-known command consoles which allows the execution of console commands like a Maven build.

Used versions

We used the current versions of the plug-ins provided by the update sites.

8.1.2.2 Target Platform

Overview

A simplification for your bundle development provides the definition of a target platform. Other than using the default target platform provided by Eclipse it's a clean platform. The installed bundles provide only the base functionality, like OSGi framework or Eclipse RCP.

It forces you to manage all of the referenced bundles clearly, because the platform contains not all possible bundles which are already installed in your Eclipse.

Setup

To create a new target platform follow the instructions below:

1. Download the Equinox OSGi runtime environment as your base target platform from the Eclipse website.
2. Download additionally the Eclipse Platform (RCP) and its Delta Package from the same website.
3. Extract the downloaded runtime environment, platform & the delta package.
4. Download additional bundles for your target platform, like the Spring Dynamic Modules.
5. Add them to the other plug-ins of the target platform.
6. Now you have setup your target platform, now we have to add it, so we can use it for our bundle development.

7. To do this open Eclipse.
8. Choose the menu *File* > *New* > *Other…*
9. Now, choose *General* > *Project* and click *Next*.
10. Enter a name of your choice and click *Finish*.
11. Select the new project and right click.
12. Choose *File* > *New* > *Other…*
13. Select *Plug-in Development* > *Target Definition* and click *Next*.
14. Enter a *file name* for your target platform configuration and click *Finish*.
15. Now when you open the target platform configuration, you can see a view like the following one.

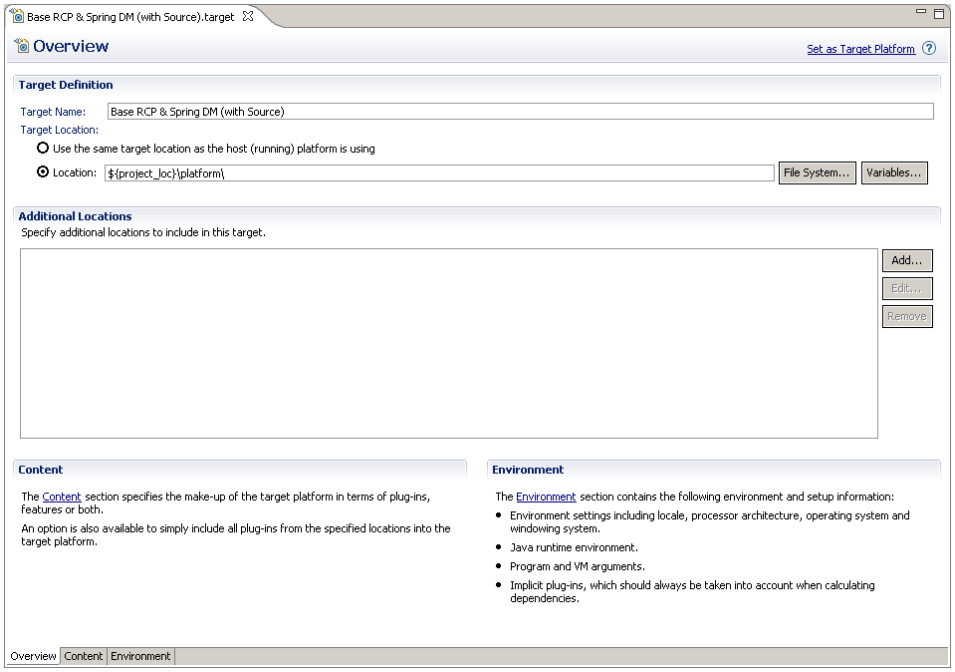

Figure 33: Target Platform file view in Eclipse

16. Enter the *Target Name*.
17. Now you have to choose the location of your target platform. To do this, select the option *Location* and click *File System…* and browse.
18. If you want to add additional bundles which are not located in the target platform you can do this just below the *Location*.
19. You can take a look through the other options and settings, but we don't want to set them right now. So, save your changes.
20. Now we created the target platform definition, but we still have to configure Eclipse to use it. We do this in the preferences.
21. Open the Eclipse preferences.
22. Choose in the left tree *Plug-in Development* > *Target Platform*.
23. In the lower part you can see *Pre-defined Targets* like the **Figure 34** shows.

24. Click on the *Browse* button next to the drop down selection and choose the target platform configuration file you've create before.
25. Click on *Load Target* to load the target with its bundles. You will see, the displayed plug-ins in the center of the *Target Platform* preferences will change as well as the *Location* shown in the upper part.
26. Check if all plug-ins are selected.
27. Now we configured the *Target Platform* for Eclipse. Click *Apply* and close the preferences.

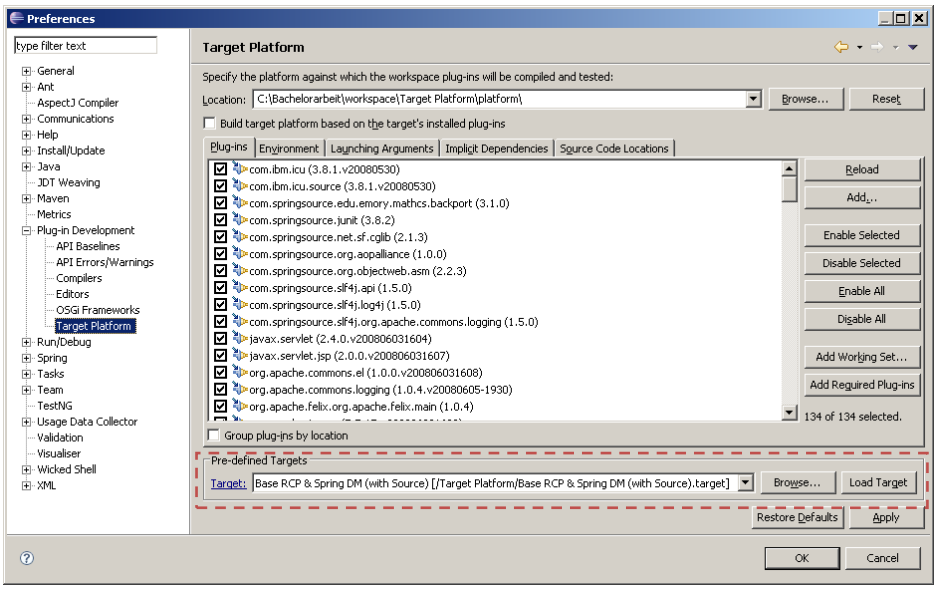

Figure 34: Target Platform configuration in the Eclipse Preferences

Extension of the target platform

During further development steps it's possible to extend the target platform with further plug-ins and bundles like GEF, EMF, GMF or others. This can be achieved easily through the addition of the plug-ins and features to the existing target platform and reload in the preferences dialog shown in **Figure 34**.

So we recommend you to consider the different versions of plug-ins, because mostly they have dependencies to other plug-ins with specific version ranges. This is a very important issue when you want to extend the target platform with further Eclipse plug-ins. This can be necessary, because the RCP Runtime contains only a minimal amount of plug-ins. Some providing for example a TextEditor or IDE views are not available, why they have to be copied by hand from your Eclipse distribution. To avoid compatibility problems take the plug-ins from the same Eclipse distribution version you use the RCP Runtime from.

In our case, we added our target platform with the plug-ins and features listed below:

- The plug-ins providing the *TextEditor* features including its source for an easier development. This covers `org.eclipse.jface.text`, `org.eclipse.text`, `org.eclipse.ui.editors`, `org.eclipse.ui.views` and `org.eclipse.ui.workbench.texteditor`.
- The `org.eclipse.core.filesystem` and `org.eclipse.core.net` plug-ins, because they were missing in the RCP Runtime.
- The Springframework Util (`org.eclipse.springframework.util`)

provided by Martin Lippert.

- The GEF runtime to allow the implementation of GEF diagrams.

Use an existing target platform

For this bachelor thesis a specific target platform was created and checked-in the SVN source management. Therefore, future users don't have to setup an own target platform, they simply can check it out and add it to their Eclipse.

After the check-out they can follow the instructions above from step 20.

Used versions

The platform used for this bachelor thesis consisted of the Eclipse Equinox and Eclipse Platform (RCP) in the version 3.4 (Ganymede) completed with the Spring Dynamic Module bundles in version 1.1.3, the Springframework Utilities provided by Martin Lippert *[Lippert08]* and related libraries.

8.1.2.2.1 Hints

Compact target platform

We recommend you to create a small and compact target platform only with the base bundles. Additional bundles and libraries you might need in the future can be added later or in combination with the Maven build and deployment process.

The compact target platform allows you additionally to exchange the target platform within a short time not affecting the bundles and libraries used in your own bundles.

Target platform sharing

Additionally, we recommend you to add the target platform in a folder of your target platform project in Eclipse. You can share the project including the target platform easily with the other team members through a source management tool like CVS or SVN. In this case you have to add a relative path in the Location of the platform using the eclipse variable `${project_loc}`.

8.1.3 Maven

Overview

For a more comfortable development, the usage of a build tool is very popular. Maven is one of the most favorite and flexible tools supporting the developer with the dependency management. Additionally, Maven can be extended with plug-ins for generating the Manifest as well as building and deploying your bundles.

Setup

To install Maven in a proper manner follow the instructions below:

1. Download the current Maven version.
2. Extract it to a directory of your choice.
3. Open the system properties and chose the tab *Advanced*.
4. Open the *Environment Variables*.
5. Add a new entry MAVEN_HOME to the directory where you installed Maven in step 2.
6. Close the Environment Variables and the system properties.

7. Now you have to create your local maven settings. To this, go the folder of your user account. Normally it's some kind of *C:\Documents and Settings\«your user»*.
8. Create a new folder called *.m2*.
9. Now you only have to create a file called settings.xml in the new folder and add the following content.

```
<settings>
    <localRepository>
      «path to your local repository»
    </localRepository>
    <mirrors>
        <mirror>
            <id>«id of the central repository»</id>
            <name>
                «name of the central repository»
            </name>
            <url>
                «url to the central repository»
            </url>
            <mirrorOf>central</mirrorOf>
        </mirror>
..  </mirrors>
</settings>
```

10. Open the Command Prompt.
11. Enter the command mvn -version and click enter.
12. Now you should see a result like this

```
Maven version: 2.0.9
Java version: 1.5.0_17
OS name: "windows xp" version: "5.1" arch: "x86" Fam-
ily: "windows"
```

Bundle Plugin for Maven

After the installation of Maven on your computer, you can build and deploy standard Java applications but no bundles. Additionally, you need the *Bundle Plug-in for Maven* from the Apache Felix project. It supports you with the generation of the Manifest file, the building and deployment of bundles for your target platform.

To use this plug-in, you only have to add the following snippet either in the *pom.xml* of your parent project or in each *pom.xml* of your projects.

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.0.0</version>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      ...
    </instructions>
  </configuration>
</plugin>
```

Listing 127: Snippet of the Bundle Plug-in for Maven in the pom.xml

Further, you have to change the value of the *packaging* tag from jar to bundle, like shown below.

```
<packaging>jar</packaging>

<packaging>bundle</packaging>
```

Listing 128: Snippet of the packaging tag

Used versions

For this bachelor thesis, we used the Maven version 2.0.9. The Bundle Plug-in for Maven was downloaded directly from the Apache Felix website through Maven why we used the most current version.

8.1.4 Libraries

Overview

The functionality of the target platform consisting of the OSGi core and the Eclipse RCP bundles can be extended with the integration of additional libraries or own developed bundles and plug-ins.

Spring

An important extension of the basic target platform is the Spring Dynamic Module with its dependencies, which allows the usage of OSGi simplifications and the usual Spring functionalities.

We added the Spring DM and the Spring libraries to the target platform from scratch, because the implementations of the future bundle will all use the Spring functionalities. That's why it is a basic infrastructure for this bachelor thesis.

8.1.5 Automatic Build

Overview

The automatic build is an important part of a development environment. It supports you with the automation of repetitive work, like the management of dependencies, builds of the whole application, running tests and further more. This work has to be done all by hand if no automatic build is available which can do these things by the way for you.

In this chapter we show you how you can setup your own automatic build environment using *Maven*, the *Apache Felix Bundle Plug-in for Maven* and the *PDE Build*. We give you advices and recommendations based on books, the internet websites and our experiences.

Structure

As mentioned in the chapter 6.1.1 of our solution, the automatic build consists of the several components, which covers specialized parts of the whole process, like the manifest generation. The following ones are the key components of the automatic build process.

- Maven build with its dependency management
- Maven Bundle Plug-in supporting the bundle handling
- Headless PDE Build for the product generation

Although the components are based on different technologies and can exist independent from the others, they interleave and complement each other to build a powerful integrated build environment.

Testing

A component not mentioned until now is the test support. We recommend you to use the Spring DM testing framework, because it allows with its utilities and simplifications a comfortable implementation of unit and integration tests. So we don't treat the testing topic in this chapter and refer to chapter 8.6.

8.1.5.1 Maven & Bundle Plug-in

Definition

The Maven build tool builds the base of our build environment. While the Maven build tool provides the well-known utilities to compile and build Java archives as well as the dependency management, the *Bundle Plug-in for Maven* relies on the former and extends it with the bundle development support. As a result of the strong integration of the plug-in in Maven we treat both topics together in this chapter and not separately.

8.1.5.1.1 Project structure

Overview

In a first step we define the project structure of the application including the Maven build tool as the core of the automatic build process. This is necessary to allow a trouble-free setup and use of the build process.

Recommendation

So we recommend you to choose a project structure with a parent project defining common configurations and resources and child projects with the different parts of your application. The following **Figure 35** shows such structure schematically.

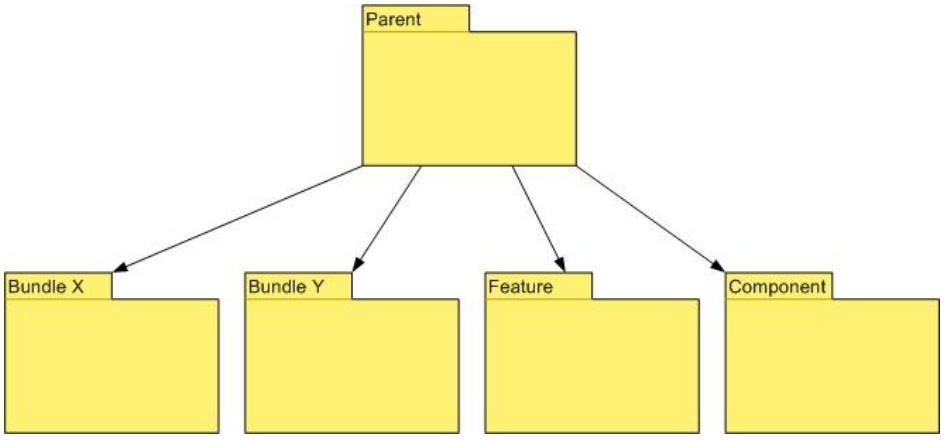

Figure 35: Recommended project structure for the automatic build

This structure has proved in the tutorials we developed during this bachelor thesis as well as several other projects, because it's the easiest way to build a whole application with all its components (bundles, fragments, features, etc.) in one Maven build. Even *SpringSource* has chosen such a structure for the development of their Spring Dynamic Modules, like the projects, contained in the SpringDM archive, show.

Additionally, we can reduce the coupling between the concrete components of your application as much as possible and can used nonetheless common confi-

guration, resources and settings provided by the parent project, like properties defining the version numbers of the used bundles and libraries.

Without the definition of a parent project, which combines all the mentioned tasks, duplicated configurations would complicate the development. Additionally special scripts would be needed to run all components of the application in a single build.

It is possible to extend the project structure with additional hierarchy levels if you need some, for example as a root project for all unit and integration tests.

An example of a hierarchical project structure is explained amongst other things in the tutorial in chapter 16.

Alternatives

Real alternatives to this structure do not exist, because each other structure causes disadvantages with at least one of the used technologies and tools.

The only one alternative which is worth to be mentioned is the one with the nested projects. The child projects are contained in the *features* and *plugins* folder of the parent project. It brings also all the advantages mentioned above with it but leads to conflicts when the projects should be managed by a version control system like CVS or SVN.

The following **Figure 36** shows the alternative project structure schematically. In contrast to the recommended structure the child projects are nested in the parent project.

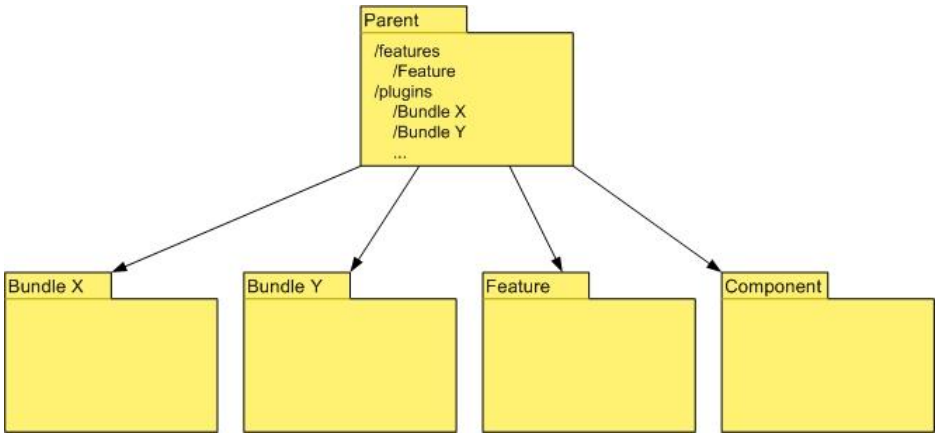

Figure 36: Alternative project structure for the automatic build

8.1.5.1.2 Bundle plug-in

Overview

Based on the project structure we defined in the chapter before, the extension of the Maven build with the plug-in supporting the bundle development is very easy to achieve.

Recommendation

So we recommend you to for the extension of the Maven build with the bundle handling the following allocation of the configurations and settings.

- Definition of the *Bundle Plug-in for Maven* in the parent project for a joint use.
- Definition of properties for the manifest generation in the parent project and its initialization with default values.

- Definition of the packaging type *bundle* in the child projects which represents a bundle or plug-in.
- Override the required properties for the manifest generation in the child projects.

Plug-in definition

The definition of the *Bundle Plug-in for Maven* for its use in the build process can be achieved with a few steps. We recommend you to locate it in the parent project for a joint use in the child projects. But the plug-in is only executed for projects which have at its disposal the packaging type *bundle*. For others, the plug-in will be ignored.

So we could reduce the duplicate definition of the plug-in in each child through a central one. This leads to a reduced configuration effort for you and a uniform manifest configuration for all bundles and plug-ins. Additionally, it allows a more stable adaptation and extension in further development steps for all bundles and plug-ins.

The following snippet shows how the plug-in have to be added in the pom of the parent project. How you can see, it is defined like all other plug-ins used to extend the Maven build.

```
<plugins>
  ...
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>2.0.0</version>
    <extensions>true</extensions>
    <configuration>
      <manifestLocation>META-INF</manifestLocation>
      <instructions>
        <Bundle-Name>
          ${artifactId}
        </Bundle-Name>
        <Bundle-SymbolicName>
          ${manifest.symbolicname}
        </Bundle-SymbolicName>
        <Import-Package>
          ${manifest.import}
        </Import-Package>
        <Export-Package>
          ${manifest.export}
        </Export-Package>
        <Private-Package>
          ${manifest.private}
        </Private-Package>
        <Bundle-Activator>
          ${manifest.activator}
        </Bundle-Activator>
        <Require-Bundle>
          ${manifest.require}
        </Require-Bundle>
        <Include-Resource>
          ${manifest.resource}
        </Include-Resource>
        <Bundle-RequiredExecutionEnvironment>
          ${manifest.environment}
        </Bundle-RequiredExecutionEnvironment>
        <Bundle-ActivationPolicy>
```

```
          ${manifest.activationPolicy}
        </Bundle-ActivationPolicy>
      </instructions>
    </configuration>
  </plugin>
  ...
</plugins>
```

Listing 129: Definition of the Bundle Plug-in for Maven in the pom file of the parent project

The tags used between the `instructions` are special. They are used for the automatic manifest generation during the build process. Because of this feature, the manual creation and adaptation can be dropped.

The names of the tags are mapped one to one as the entry names in the manifest and the values as the values of the corresponding entry in the manifest. The tags are not defined in a schema, which allows the use of all kinds of tags.

Properties

For the values in the configuration of the manifest generation, we used properties as placeholder for its values. The use of them was a natural consequence of the plug-in definition in the parent project, because the concrete values can only be set in the child projects.

Additionally, the definition of the properties in the pom file of the parent project offers advantages. You can initialize them with default values which are used when the properties are not overwritten.

```
<properties>
  <manifest.symbolicname>
    ${project.groupId}.${project.artifactId}
  </manifest.symbolicname>
  <manifest.private></manifest.private>
  <manifest.export></manifest.export>
  <manifest.import>
    !${manifest.export},
    *;resolution:=optional
  </manifest.import>
  <manifest.activator></manifest.activator>
  <manifest.require></manifest.require>
  <manifest.resource></manifest.resource>
  <manifest.classpath>.</manifest.classpath>
  <manifest.environment>J2SE-1.5</manifest.environment>
  <manifest.activationPolicy></manifest.activationPolicy>

  <!-- Other properties -->
  ...
</properties>
```

Listing 130: Definition of the properties for the manifest generation in the pom file

The properties can be defined in the `properties` tag directly which is located in the `project` tag. The names and values are free of choice.

Bundle and plug-in projects

The configurations we discussed until now were all located in the pom file of the parent project. They define common parts of the whole bundle handling in the build process.

But we also have to do some configuration steps in the child project which represents the bundles in the whole application. First we begin with the change

of the packaging type to *bundle*. Otherwise the *Bundle Plug-in for Maven* will not be executed.

```
<packaging>bundle</packaging>
```

Listing 131: Definition of the packaging type in the pom file

Additionally, we have to override the properties for the manifest generation. You can do this in the same we also defined it in the parent project but with the difference you only have to use them which are really required.

```
<properties>
  <manifest.symbolicname>
    ${project.groupId}.${project.artifactId};singleton:=true
  </manifest.symbolicname>
  <manifest.export>
    org.herasaf.tutorial.calculator.services;
        version=${project.version}
  </manifest.export>
</properties>
```

Listing 132: Override of the properties for the manifest generation in the pom file

That's all you have to do for the integration of the bundle handling in the existing Maven build.

Alternatives

Beside the different plug-ins and tools already described in chapter 6.1.1.2.1 a further alternative exists. Instead of defining the *Bundle Plug-in for Maven* in the parent project and the corresponding properties, you can also define it in the concrete child projects where you need it.

On the whole, it is the same solution as the recommended one, but with the problem of duplicate configurations and the consequent disadvantages. This appears especially in applications which are based on several bundles and plug-ins. So we don't recommend you to integrate the bundle handling in this way.

8.1.5.1.3 Hints

Additional hierarchy level

If you want to add configurations, plug-ins and other resources to the pom file of your parent, which doesn't apply for all child projects, extend your project structure with an additional hierarchy level. With this step you can preserve the flexibility of the project structure and define nonetheless specific configurations for only some of the child projects.

Sequence of the child projects

Please consider the sequence of the child projects in the parent. You have to assure that all dependencies for each project are compiled and built. You can achieve this goal if you choose a bottom up build order, based on defined the architecture.

With this considerations can avoid unnecessary exceptions and errors during the build.

Maven Class-

If you create a plug-in project in Eclipse, you should remove in a first step the

path Container

Maven Classpath Container. Otherwise you will get conflicts with the *Plug-in Dependencies* due to duplicate project entries in the class path and further more. The cause of this problem is the missing compatibility between the Maven plug-in and the Eclipse Plug-in Development Environment.

Manifest generation

We notice from time to time a wrong or incomplete generation of the manifest file during the build. If such a problem occurs in your project, try to delete the existing manifest file, because the plug-in seems to have problems with overwriting existing manifests.

8.1.5.2 PDE Build

Definition

The build we setup until now handles only the build of bundles and plug-ins in your application in conjunction with the dependency management with the repositories.

The PDE Build we are going to integrate now is an extension of the existing build with the ability to create executable products. It is not another Maven plug-in which allows such a product creation, but a component provided by the Eclipse PDE distribution.

Precondition

For that reason, you need an installed **Eclipse PDE** on your workstation to run the PDE Build. Otherwise you will get errors.

Additionally, you need a **target platform** you can run the build against it. This can be your Eclipse installation or a self defined target platform like described in chapter 8.1.2.2.

8.1.5.2.1 Build project

Recommendation

We recommend you to create an own project for the integration of the Headless PDE Build in your build process. The reason behind this idea is to define generic Ant build scripts you can use in all of your other projects, where you want to build products. The configuration with specific behavior and settings is achieved by properties which are passed to the scripts of the Headless PDE Build.

The following snippet shows such a generic Ant build script, which carries out preparations before executing the PDE Build and post-processing steps after the build.

```
<?xml version="1.0"?>
<!--
    This program and the accompanying materials are made
    available under the terms of the Eclipse Public License v1.0
    which accompanies this distribution, and is available at
      http://www.eclipse.org/legal/epl-v10.html

    This build script creates a build directory containing the
    plugins and features to be built, and then kicks off the PDE
    build process. You could just as easily do this from a shell
    script or cron job.

    Also, the script can be run inside the Eclipse IDE by
```

```
    Choosing Run As -> Ant Build from the context menu. It could
    obviously be run outside of the IDE if you have ANT installed
    on your path.
-->
<project default="build">
  <property file="build.properties" />
  <!--
        PDE Build expects that the build directory contains a
        "plugins" directory and a "features" directory. These
        directories should contain the various plug-ins and
        features to be built.

        It's possible to use the CVS checkout process that is
        built into PDE Build. This is done with map files and is
        beyond the scope of this tutorial.

        This tutorial simply copies the projects directly from
        your workspace into the appropriate build directory
        folders.
    -->
  <target name="init">
    <mkdir dir="${buildDirectory}" />
    <mkdir dir="${buildDirectory}/plugins" />
    <mkdir dir="${buildDirectory}/features" />

    <echo message="${projectPrefix}" />

    <copy todir="${buildDirectory}/plugins">
      <fileset dir="../">
        <include name="${projectPrefix}.*/**" />
        <exclude name="${projectPrefix}*feature*/**" />
      </fileset>
    </copy>
    <copy todir="${buildDirectory}/features">
      <fileset dir="../">
        <include name="${projectPrefix}*feature/**" />
      </fileset>
    </copy>
  </target>

  <!--
        This target actually executes the PDE Build process by
        launching the Eclipse antRunner application.
    -->
  <target name="pde-build">
    <echo message="Executing ${eclipseLocation}/plugins/
           org.eclipse.equinox.launcher_
           ${equinoxLauncherPluginVersion}.jar" />
    <java classname="org.eclipse.equinox.launcher.Main"
          fork="true" failonerror="true">
      <arg value="-application" />
      <arg value="org.eclipse.ant.core.antRunner" />
      <arg value="-buildfile" />
      <arg value="${eclipseLocation}/plugins/org.eclipse.
                  pde.build_${pdeBuildPluginVersion}
                  /scripts/productBuild/productBuild.xml" />
      <arg value="-Dtimestamp=${timestamp}" />
      <arg value="-DallowBinaryCycles=true" />

      <!-- Properties set in the global or project settings -->
      <arg value="-DbuildDirectory=${buildDirectory}" />
      <arg value="-DeclipseLocation=${eclipseLocation}" />
      <arg value="-DequinoxLauncherPluginVersion=
                  ${equinoxLauncherPluginVersion}" />
      <arg value="-DpdeBuildPluginVersion=
```

```
                    ${pdeBuildPluginVersion}" />
      <arg value="-DtopLevelElementType=${topLevelElementType}"/>
      <arg value="-DtopLevelElementId=${topLevelElementId}" />
      <arg value="-Dproduct=${product}" />
      <arg value="-DarchivePrefix=${archivePrefix}" />
      <arg value="-Dconfigs=${configs}" />
      <arg value="-DbuildType=${buildType}" />
      <arg value="-DbuildId=${buildId}" />
      <arg value="-Dbase=${base}" />
      <arg value="-DbaseLocation=${baseLocation}" />
      <arg value="-Dbaseos=${baseos}" />
      <arg value="-Dbasews=${basews}" />
      <arg value="-Dbasearch=${basearch}" />
      <arg value="-DjavacSource=${javacSource}" />
      <arg value="-DjavacTarget=${javacTarget}" />
      <arg value="-DprojectPrefix=${projectPrefix}" />

      <classpath>
        <pathelement location="${eclipseLocation}/plugins/org.
          eclipse.equinox.launcher_
          ${equinoxLauncherPluginVersion}.jar" />
        </classpath>
    </java>
  </target>

  <!--
        This target actually cleans the build directory after the
        build.
    -->
  <target name="post-build-cleanup">
    <delete includeemptydirs="true">
      <fileset dir="${buildDirectory}">
        <exclude name="${buildLabel}/*.zip" />
      </fileset>
    </delete>
  </target>

  <target name="clean">
    <delete dir="${buildDirectory}" />
  </target>

  <target name="build" depends="init, pde-build, post-build-
cleanup" />
</project>
```

Listing 133: Example of a Ant build script for the Headless PDE Build

In the example above, you can see, we copy the projects to a special build directory to avoid the direct build in the project folders. We have to do this, because the PDE Build, provided by the Eclipse PDE distribution, requires a defined folder structure, shown in the following listing. Additionally, the PDE differentiate between bundles/plug-ins and features, because the latter don't have to be compiled. So we have to create this structure and copy our projects into them, before we can execute the PDE Build. We recommend you to do it the same way.

```
/«build path»
  /features
  /plugins
```

Listing 134: Folder structure required for the PDE Build

After the successfully executed PDE Build, we clean the build directory from the copied projects, generated scripts and other contents, to leave only the generated archive there.

Alternatives

This is the solution we recommend you to use, because it's the most flexible one and reduces duplicate code and configuration in most. But there's also an alternative solution where you add the build script for the Headless PDE Build in one of the projects of your application.

It works like the solution located in an own project, but leads to additional advantages and disadvantages.

One of the advantages is that you can add the project specific properties in the *build.properties* file which accompanies the *build.xml* file. You don't have to define them in the pom.xml anymore, because the Headless PDE Build is already project specific. Additionally, you don't have to pass the project specific properties from your build script to the PDE Build provided by Eclipse, because those scripts read the *build.properties* file in the working directory.

The location and passing of the infrastructure specific properties remains the same, like in the recommended solution.

An enormous disadvantage of this solution is the duplication of the build script (*build.xml*) for each application and correspondingly the *build.properties* with the general properties which would apply for all applications, not only the one where it is contained in.

So we recommend you this solution only in cases where you create a single application and don't want to manage a further project for the Headless PDE Build.

8.1.5.2.2 Build properties

Motivation

The properties in the Headless PDE Build are used for the configuration of the whole PDE Build. In a first step, some are used for the Headless PDE Build itself, especially for the preparation and post-processing tasks we want to provide. In a second step, they will be used to configure the PDE Build provided by Eclipse. Otherwise the plug-in would not work well or use default values which are not appropriate.

Recommendation

Based on the recommendation in chapter 8.1.5.2.1, we recommend you to split the properties in three different parts for an independent location of them.

The idea behind this recommendation is to locate the properties based on their scope. It's inappropriate to locate properties describing the infrastructure in the *build.properties*, because they differ from workstation to workstation. Also the definition of the general properties, which apply for all kinds of PDE Build, in a project specific manner, is not a proper solution, because it leads to unnecessary duplications.

So we recommend you to split the properties in the following three parts.

- General properties
- Infrastructure properties

- Project properties

General properties

The general properties should be located in the *build.properties* file, which accompanies the Headless PDE Build script (*build.xml*). The contained properties apply for all kinds of applications which use the Headless PDE Build, so it's the obvious solution to locate them beside the script. Other locations than this one results in duplications, if they are copied in each project or located somewhere else.

The following table shows the properties which should be defined as general properties. They specify mostly runtime configurations for the PDE Build and use infrastructure- or project-specific properties to combine them.

Property	Description	Default value
runPackager	Set if the packager needs to be run. Set this if your build results need to contain binary features and plug-ins that come from the *baseLocation*.	true
collectingFolder	The location under which all of the build output will be collected.	${archivePrefix}
zipargs	Arguments to send to the zip executable.	
tarargs	Arguments to send to the tar executable.	
buildLabel	Label for the build. Used in naming the build output.	${buildType}. ${buildId}
timestamp	Timestamp for the build. Used in naming the build output.	007
generateFeature VersionSuffix	Enable / disable the generation of a suffix for the features that use *.qualifier*. The generated suffix is computed according to the content of the feature.	true
filteredDependency Check	This property indicates whether you want the set of plug-ins and features to be considered during the build to be limited to the ones reachable from the features / plugins being built.	false
resolution.devMode	This property indicates whether the resolution should be done in development mode (i.e. ignore multiple bundles with singletons).	false
logExtension	Specify the output format of the compiler log when Eclipse JDT is used.	.log
javacDebugInfo	Whether or not to include debug	false

	info in the output jars.	
javacFailOnError	Whether or not to fail the build if there are compiler errors.	true
javacVerbose	Enable or disable verbose mode of the compiler.	true

Table 1: General properties for the Headless PDE Build

There exist further properties which allow the configuration of CVS fetches with the corresponding file mapping and the declaration of the Eclipse download site to mention only some of them. We skip them all through the corresponding skip properties (`skipBase`, `skipMaps`, `skipFetch`, etc.).

The following snippet shows an example of the general properties defined in the *build.properties* file.

```
############# PRODUCT/PACKAGING CONTROL #############
runPackager=true

# The location underwhich all of the build output will be col-
lected.
collectingFolder=${archivePrefix}

#Arguments to send to the zip executable
zipargs=

#Arguments to send to the tar executable
tarargs=

############## BUILD NAMING CONTROL #################

# Label for the build.  Used in naming the build output
buildLabel=${buildType}.${buildId}

# Timestamp for the build.  Used in naming the build output
timestamp=007

#Enable / disable the generation of a suffix for the features that
use .qualifier. The generated suffix is computed according to the
content of the feature.
generateFeatureVersionSuffix=true


############# BASE CONTROL #############
#This property indicates whether you want the set of plug-ins and
features to be considered during the build to be limited to the
ones reachable from the features / plugins being built
filteredDependencyCheck=false

#This property indicates whether the resolution should be done in
development mode (i.e. ignore multiple bundles with singletons)
resolution.devMode=false

skipBase=true
eclipseURL=<url for eclipse download site>
eclipseBuildId=<Id of Eclipse build to get>
eclipseBaseURL=${eclipseURL}/eclipse-platform-${eclipseBuildId}-
win32.zip


############# MAP FILE CONTROL #################
```

```
# This section defines CVS tags to use when fetching the map files
from the repository.
# If you want to fetch the map file from repository / location,
change the getMapFiles target in the customTargets.xml

skipMaps=true
mapsRepo=:pserver:anonymous@example.com/path/to/repo
mapsRoot=path/to/maps
mapsCheckoutTag=HEAD

#tagMaps=true
mapsTagTag=v${buildId}


############ REPOSITORY CONTROL ###############
# This section defines properties parameterizing the repositories
where plugins, fragments
# bundles and features are being obtained from.

# The tags to use when fetching elements to build.
# By default thebuilder will use whatever is in the maps.
# This value takes the form of a comma separated list of reposito-
ry identifier (like used in the map files) and the
# overriding value
# For example fetchTag=CVS=HEAD, SVN=v20050101
# fetchTag=HEAD
skipFetch=true


############# JAVA COMPILER OPTIONS ##############
# Specify the output format of the compiler log when eclipse jdt
is used
logExtension=.log

# Whether or not to include debug info in the output jars
javacDebugInfo=false

# Whether or not to fail the build if there are compiler errors
javacFailOnError=true

# Enable or disable verbose mode of the compiler
javacVerbose=true

# Extra arguments for the compiler. These are specific to the java
compiler being used.
#compilerArg=
```

Listing 135: Snippet of a build.properties accompanying the build.xml

Infrastructure-specific properties

Different than the general properties are the infrastructure-based ones. They apply also for all kinds of applications which use the Headless PDE Build, but they depend directly on the workstation, the PDE build runs on.

An obvious location for these properties is the *settings.xml* file, because it contains already user- and workstation-specific settings for Maven. These are partly infrastructure settings. So we recommend you to add this kind of properties in the Maven settings file. Additionally, this location allows an easy access of the properties without any special preparation, includes or something else.

The following table shows the properties which are infrastructure-based. They specify mostly several paths and version numbers for the correct location of versioned elements.

Property	Description	Default value
buildDirectory	The directory into which the build elements are fetched and where the build takes place.	None
eclipseLocation	The root directory of your local Eclipse PDE installation.	None
equinoxLauncher PluginVersion	Version number of the plug-in `org.eclipse.equinox.launcher`.	None
pdeBuildPlugin Version	Version number of the plug-in `org.eclipse.pde.build`.	None
targetPlatforms Directory	The location, where the build process can find the target platforms against which the implementation should be built.	None

Table 2: Infrastructure specific properties for the Headless PDE Build

These five properties are the ones we used in our implementation of the Headless PDE Build. We recommend you also to use them, but feel free to insert additional ones if you need them.

The following snippet shows an excerpt of the *settings.xml* file, which contains configurations and settings for the Maven build environment, like the path to the local Maven repository.

```
<settings>
  ...
  <profiles>
    ...
    <profile>
      <id>headlessPDEBuild</id>
      <properties>
        <buildDirectory>
          C:/temp</buildDirectory>
        <eclipseLocation>
          C:/Programme/eclipse
        </eclipseLocation>
        <!-- Version number for org.eclipse.equinox.launcher -->
        <equinoxLauncherPluginVersion>
          1.0.101.R34x_v20081125
        </equinoxLauncherPluginVersion>
        <!-- Version number for org.eclipse.pde.build -->
        <pdeBuildPluginVersion>
          3.4.1.R34x_v20081217
        </pdeBuildPluginVersion>
        <targetPlatformsDirectory>
          ${basedir}/../TargetPlatforms
        </targetPlatformsDirectory>
      </properties>
    </profile>
  </profiles>
```

```
  <activeProfiles>
    <activeProfile>headlessPDEBuild</activeProfile>
    ...
  </activeProfiles>
</settings>
```

Listing 136: Snippet of a infrastructure-based properties definition in the settings.xml

Project-specific properties

After the general and the infrastructure-based properties, we only have to define the project-specific ones. They define the remaining PDE Build configurations which aren't part of the other kind of properties. Examples are the location of the *.product* file, which platform exports should be supported or the location of the target platform.

The most obvious location for them is a file in a project of your application, which should be built. We recommend you to add them in the *pom.xml* file of the project, from where you invoke the Headless PDE Build (see chapter 8.1.5.2.3). To locate them in this file allows you to pass them in an easy way as parameters to the invoked build script. To define them you can also use already defined Maven properties.

The following table shows the project-specific properties specifying locations and configurations specific for a concrete application to build.

Property	Description	Default value
topLevelElement Type	The type of the top level element we are building, generally *feature*.	feature
topLevelElementId	The id of the top level element we are building.	
product	Location of the *.product* file, specifying the product.	${basedir}/«…»/ «…».product
archivePrefix	The prefix that will be used in the generated archive.	«name»
configs	The list of {os, ws, arch} configurations to build. This value is a '&' separated list of ',' separate triples. For example: `configs=win32,win32,x86 & linux,motif,x86` By default the value is *,*,*	
buildType	Type of build. Used in naming the build output. Typically this value is one of I, N, M, S, ...	I
buildId	ID of the build. Used in naming the build output.	«name»
base	Base location for anything the build needs to compile against. For example, in most RCP app or a plug-in, the *baseLocation* should be the location of a previously installed Eclipse against which the applica-	${targetPlatforms Directory}
baseLocation		${base}/«platform»

	tion or plug-in code will be compiled and the RCP delta pack.
baseos basews basearch	Os/Ws/Arch/nl of the eclipse specified by *baseLocation*.
javacSource	Default value for the version of the source code. This value is used when compiling plug-ins that do not set the *Bundle-RequiredExecutionEnvironment* or set *javacSource* in *build.properties*
javacTarget	Default value for the version of the byte code targeted. This value is used when compiling plug-ins that do not set the *Bundle-RequiredExecutionEnvironment* or set *javacTarget* in *build.properties*.
projectPrefix	Prefix of the projects, which should be integrated in the PDE build.

Table 3: Project specific properties for the Headless PDE Build

The property `projectPrefix` brings a restriction with it, because it forces you to name the projects of your application all in the same manner and with the same name prefix. Otherwise the Headless PDE Build is not able to determine which projects in the same folder, mostly the Eclipse workspace, belongs to the application you want to build.

For example, if you implemented a calculator application with the following projects, which are bundles, features and other ones, the property forces you to name them all with the `org.herasaf.tutorial.calculator` prefix to determine their affiliation.

- org.herasaf.tutorial.calculator
- org.herasaf.tutorial.calculator.ui
- org.herasaf.tutorial.calculator.core
- org.herasaf.tutorial.calculator.feature

The following code snippet shows an excerpt of the *pom.xml* file, contained in the project which invokes the Headless PDE Build. It defines the last category of properties which are neither infrastructure-specific nor general applicable in the `tasks` tag of the Maven AntRun Plug-in (see chapter 8.1.5.2.3).

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    ...
    <!-- Execute the product build -->
    <execution>
      ...
      <configuration>
        <tasks>
```

```
            ...
            <!-- Project settings -->
            <property name="topLevelElementType" value="feature" />
            <property name="topLevelElementId" value="Calculator" />
            <property name="product"
                      value="${basedir}/../
                             org.herasaf.tutorial.calculator.ui/
                             calculator.product" />
            <property name="archivePrefix" value="calculator" />
            <property name="configs" value="win32, win32, x86" />
            <property name="buildType" value="I" />
            <property name="buildId" value="Calculator" />
            <property name="base"
                      value="${targetPlatformsDirectory}" />
            <property name="baseLocation"
                      value="${base}/equinox-3.4.2" />
            <property name="baseos" value="win32" />
            <property name="basews" value="win32" />
            <property name="basearch" value="x86" />
            <property name="javacSource" value="1.5" />
            <property name="javacTarget" value="1.5" />
            <property name="projectPrefix"
                      value="org.herasaf.tutorial.calculator" />

            ...
          </tasks>
        </configuration>
      </execution>
    </executions>
</plugin>
```

Listing 137: Snippet of the projects-specific properties in the pom.xml of the invoking project

For the definition of the project-specific properties in the *pom.xml* file of the invoking project you can use already existing Maven properties. This allows a more seamless integration in the build process.

Alternatives

A real alternative for the recommended implementation of the Headless PDE Build exists on in a limited manner.

If you have only one application you will build with a PDE Build, you can integrate it also in you project. This has the advantage of simple properties configuration, because only the infrastructure-specific ones have to be outsourced. But this is the only advantage and so we don't recommend you this alternative.

8.1.5.2.3 Integration

Recommendation

After the definition of the project for the Headless PDE Build and the properties for its configuration, we have to integrate it in the existing build process. In our case this is the Maven build.

AntRun Plug-in

Since we defined the Headless PDE Build as an Ant script, we need a possibility to invoke it from Maven. The Maven AntRun Plug-in allows you the implementation of conventional Ant tasks in Maven. So we recommend you to use it for the invocation of the Headless PDE Build script.

The following code snippet shows how to add the Maven AntRun Plug-in to Maven.

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <!-- Clean the product build directory -->
    <execution>
      <id>clean-product</id>
      <phase>clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <property name="buildDirectory"
                    value="${buildDirectory}" />
          <ant dir="${basedir}/../org.herasaf.pde.pdebuild"
               target="clean" inheritRefs="true" />
        </tasks>
      </configuration>
    </execution>
    <!-- Execute the product build -->
    <execution>
      <id>build-product</id>
      <phase>install</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <!-- Global settings -->
          <property name="buildDirectory"
                    value="${buildDirectory}" />
          <property name="eclipseLocation"
                    value="${eclipseLocation}" />
          <property name="equinoxLauncherPluginVersion"
                    value="${equinoxLauncherPluginVersion}" />
          <property name="pdeBuildPluginVersion"
                    value="${pdeBuildPluginVersion}" />
          <!-- Project settings -->
          <property name="topLevelElementType"
                    value="feature" />
          <property name="topLevelElementId"
                    value="Calculator"/>
          <property name="product"
                    value="${basedir}/../
                           org.herasaf.tutorial.calculator.ui/
                           calculator.product" />
          <property name="archivePrefix"
                    value="calculator" />
          <property name="configs"
                    value="win32, win32, x86" />
          <property name="buildType" value="I" />
          <property name="buildId" value="Calculator" />
          <property name="base"
                    value="${targetPlatformsDirectory}" />
          <property name="baseLocation"
                    value="${base}/equinox-3.4.2" />
          <property name="baseos" value="win32" />
          <property name="basews" value="win32" />
          <property name="basearch" value="x86" />
          <property name="javacSource" value="1.5" />
          <property name="javacTarget" value="1.5" />
          <property name="projectPrefix"
                    value="org.herasaf.tutorial.calculator"/>
```

```
            <ant dir="${basedir}/../org.herasaf.pde.pdebuild"
                 target="build" inheritRefs="true" />
          </tasks>
        </configuration>
      </execution>
    </executions>
</plugin>
```

Listing 138: Example of the Maven AntRun Plug-in in the pom.xml of the feature project

The definition of the properties before the invocation of the Headless PDE Build script is required to provide them in the Ant script. This works, because the default value for the attribute `inheritAll` of the `ant` tag is set to *true*. This means all defined properties will be inherited. So a simple definition is enough.

Feature Project

For the location of Headless PDE Build invocation, we recommend you to create a feature project. In its *pom.xml* file you can add the plug-in for the invocation. But why a separate project? You can add the plug-in also in the *pom.xml* file of your parent project or somewhere else. It works, but the sequence of the build isn't correct anymore. If you add the invocation in the parent project, it executes the build before the bundles, fragments, features and other projects are built and tested. Similar behavior occurs when you add the invocation in another, already existing project.

So our recommendation to you is to create a feature project and to add the Headless PDE Build invocation in its *pom.xml* file. Afterwards you are able to add the feature project as the last project to build in the parent project, as shown in the following code snippet of the *pom.xml* file.

```
<modules>
  ...
  <module>../org.herasaf.tutorial.calculator.feature</module>
</modules>
```

Listing 139: Snippet out of the pom.xml of the parent project for the feature integration

Additionally, you can use the feature project for the manual export of your application as a feature, if you don't want to generate a product. So you have both possibilities either to export it as a product automatically or a feature manually with the Eclipse export function.

Alternatives

As an alternative to the recommended solution you can integrate the invocation in an existing project like the parent, but you have consider a wrong sequence of the single build steps. It works well too, but is not a clean solution, why we don't recommend doing it this way.

8.1.5.2.4 Hints

Intention

During the design and implementation of the Automatic Build, we had a lot of problems to solve and challenges to take. The most of them are listed and explained in the chapters 6.1.1.1.2, 6.1.1.2.3 and 6.1.1.3.3. Based on them we give you in this chapter hints how you can easily avoid them.

Product synchronization

During the implementation of your application, you add and remove dependencies to other bundles and fragments. When you execute the PDE Build or the generated product, you will get errors. A main cause is the product file, because it isn't up to date anymore. A regular synchronization of this file and the update of the dependencies prevent the most of the errors and exceptions.

feature.xml

Also problems can occur in cases where the feature.xml file in your feature project is not up to date anymore. Also in this case, a regular synchronization can help preventing errors and exceptions in the PDE Build or the execution of the generate product.

Run configuration

Delete also the run configuration in your Eclipse from time to time. Even if you synchronize your product regularly, sometimes the run configuration isn't up to date anymore. The deletion of it and the creation of a new one afterwards help to prevent errors during the execution of your plug-ins and applications in your Eclipse.

config.ini

An insidious problem can occur during the PDE Build, if you use bundles and plug-ins but don't without an entry in the *Require-Bundle* or *Import-Package* of the manifest. In this case, the build generates an incomplete *config.ini* file, which defines the bundles and plug-ins which should be started during the startup phase of your application. A typical example of this the use of Spring DM.

In these cases you have to define an own config.ini file with the correct number of bundles and plug-ins to start. How you can do this is explained in detail in *[Daum2008]*.

Split packages

During the implementation of RCP applications the use of split packages is inevitable, because they are used in core packages of RCP. A detailed list about the distribution of the packages among the plug-ins can be found in the Eclipse Help *[PluginDevGuide]*. An insufficient understanding can lead to problems. So we recommend you first to acquaint yourself with the split packages problem, described very well in the OSGi Service Platform Core Specification *[OSGiCore]*.

To solve problems caused through the split packages, only one solution works reliable. Use in the pom file properties for the import directive the negative pattern for the affected package. Additionally, add the bundles providing the split packages to the *Require-Bundle* entry. This directs the *Bundle Plug-in for Maven* to leave out the package during the generation of the Manifest file. In this case Eclipse and the OSGi environment will use instead the *Import-Package* directive the *Require-Bundle*, which provides the "whole" package with all classes.

```
<manifest.import>
  ...,
  !org.eclipse.core.runtime;version="3.4.0",
  org.eclipse.equinox.app,
  ...
</manifest.import>

<manifest.require>
  org.eclipse.core.runtime;bundle-version="3.4.0",
  ...
</manifest.require>
```

Manifest format-

After the generation of the Manifest file through the *Bundle Plug-in for Maven*,

ting

Eclipse may show inexplicable errors. Although all the entries and directives in the file are correct, Eclipse cannot read them correctly. The cause of the problem is the formatting of the Manifest file. If you format it with the keyboard shortcut *CTRL+SHIFT+F*, the problem solves itself. It seems that Eclipse has problems reading the generated Manifest file properly.

Unfortunately this has also direct impact on the PDE Build itself. It seems that it uses the mechanism to read the Manifest file, like the Eclipse IDE, too. For this reason it is not possible to run the build of the projects together with the Headless PDE Build.

Cycles between bundles

To prevent an additional error during the PDE Build, we recommend you to add the following parameter to your Headless PDE Build script which invokes the PDE Build provided by Eclipse. Otherwise the PDE Build will abort if he detects binary cycles between bundles. This is especially a problem when using third-party libraries, which are not under your control.

```
<arg value="-DallowBinaryCycles=true" />
```

But this is not a free ticket to implement your bundles with unnecessary dependencies. So design your application well to reduce the dependencies between your bundles.

NullPointer in the TriggerPoint Manager

An error which can occur during the PDE Build is also a `NullPointerException` in the class `TriggerPointManager`. On the face of it, no error can be recognized. The problem is the use of extension points, which are not imported.

For example, if you use the extension point `org.eclipse.ui.views`, you have to take care that you import the bundle which provides it. In this case the extension point is contained in the plug-in `org.eclispe.ui`.

Version numbers

A huge simplification, especially in large applications, brings the definition of the properties in the *pom.xml* of the parent project which contains the version numbers of the used bundles and libraries. In the derived *pom.xml* files you only have to use the properties instead the concrete version numbers.

The advantage of such a definition comes to bear particularly, when you want to upgrade to a new version of your target platform or something else. In this case you only have to change the numbers in the *pom.xml* of your parent project.

So we recommend you to use such a definition if you plan to implement a bigger application.

8.2 Architecture

Overview

The architecture with all its long-ranging decisions is an important part of each application design, because it has a direct impact on the key requirements on the one hand and the non-functional ones, described by the ISO9126 *[ISO9126]* standard.

In this chapter we show you possible solutions for different parts of an application architecture and give you advices for them based on the advantages and

disadvantages in an OSGi/RCP environment.

8.2.1 Layers

Intention

Generally, the separation of applications in different layers with its own key concerns has proven as a good solution and is therefore applied widely in the application development.

Additionally, the OSGi environment with its services supports such a layering more than other environments. Therefore a lower coupling between the layers can be achieved more easily.

Layer architecture

That's the reason why we recommend you a layering of your application like shown in **Figure 37**. Such a layering allows you to define clear dependencies between the layers which reduce the coupling to a minimum.

Additionally, it allows you to implement your application independently, layer by layer, especially if have an extensive use of interfaces. Therefore it's possible to exchange the layers easily through substitutes, which enables a finer-grained testing of your application.

Generally, such architectures are widely used in developing applications and are therefore best practice. It allows through the concentration of the key concerns the definition of clear dependencies between the layers. This increases the reusability of its classes and interfaces.

Naturally this architecture brings also disadvantages like a higher implementation and management effort, but they are only marginal compared with the huge advantages of lower coupling and an easy exchangeability of the layers.

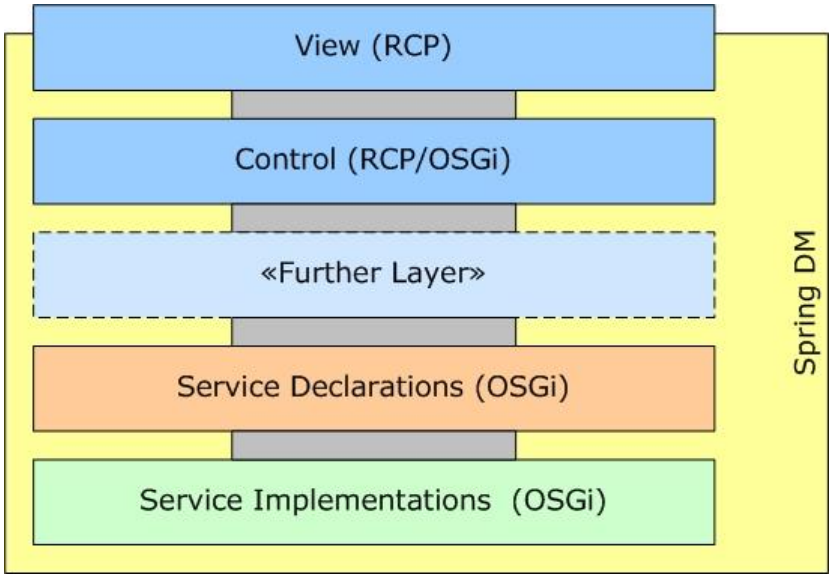

Figure 37: Proposed architecture of an RCP based application with its layers

Slice architecture

An alternative to the layering architecture is one where a bundle contains a vertical slice containing all classes and interfaces for a specific function. It replaces the horizontal layering through a "vertical" one. Unfortunately the reusability of

common parts and utilities is only hard to achieve to mention only one problem.

Therefore a real alternative is only the combination of the vertical and horizontal layering. In this case the utilities and common parts are organized in a horizontal layering architecture and the concrete components of the application in vertical slices. Such a solution allows minimizing the disadvantages of the vertical slices architecture in a considerable manner. **Figure 38** shows such a mixed architecture.

Consider the advantages and disadvantages of the different solutions to find an appropriate architecture for your application. Otherwise you will have to handle with unnecessary problems and trouble which causes extra efforts.

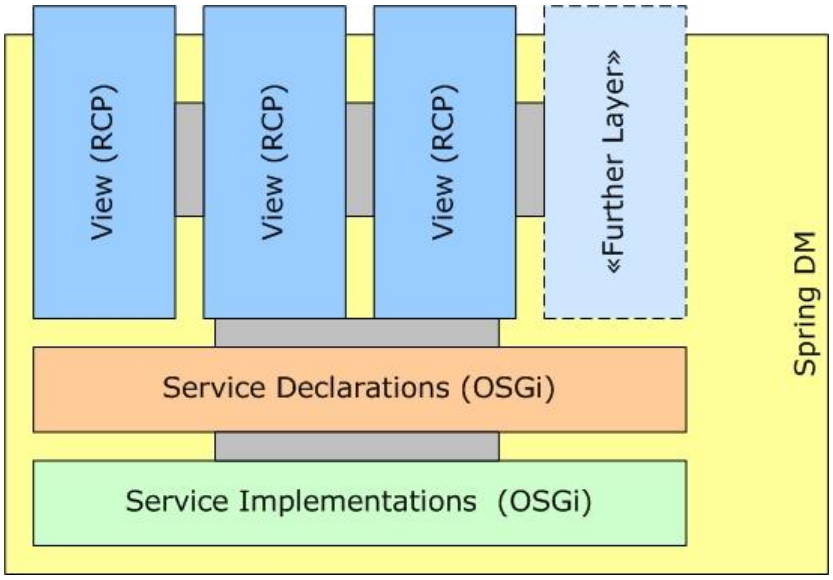

Figure 38: Alternative architecture of an RCP based application with its layers

8.2.2 General Topics

Overview

In this chapter the instructions and recommendations for the general architectural topics of the reference architecture are contained.

8.2.2.1 Bundle Naming

Overview

The naming of bundles is an important issue for its organization. This includes also the possibility to distinguish between them. A clear and consistent naming of your bundles can also help to identify its content easily.

Recommendation

Based on these requirements and own experiences, we recommend you to name your bundles using the domain name style like the following example shows. Such a name consists of several parts. The first one is the domain name of the organization which wrote respectively writes the bundle (`org.herasaf`), the second part describes the application (`tutorial.calculator`) and the last part the specific component out of the application (`ui`).

`org.herasaf.tutorial.calculator.ui`

We recommend you such a naming pattern because it allows a clear and almost unique naming of your bundles and it provides different information which enables an exact assumption of its content. Additionally, it brings advantages in combination with the package naming described in chapter 8.2.2.2.

The only disadvantage of such a naming is the relative long names compared with other naming patterns. But these provide on the other hand only a small amount of the information.

Alternatives

Naturally you can use also different pattern for naming your bundles, because our recommendation is not the only correct solution. Note you have to consider the advantages and disadvantages of each naming pattern and if it's appropriate for your project.

But we don't want to go into details about the other naming patterns, because it would break the size of this document.

8.2.2.2 Package Naming

Overview

Together with the naming of the bundles also the naming of its packages is an important issue. A clear and consistent naming of your packages in the bundles can help to reduce unnecessary problems and allow finding the classes and interfaces in an easier way.

Recommendation

Based on our recommendation in chapter 8.2.2.1 and own experiences, we recommend you to name the root package of a bundle after the bundle's name.

The reason for this recommendation is the possibility to avoid package conflicts like split packages between different bundles. This can happen if you implement classes in the same package but locate them in different bundles. With this naming pattern for the root package, such problems could be prevented.

Additionally, it allows an easy assignment of classes and interfaces to a bundle, because the root package and the bundle name correspond. If somebody searches the bundle for a class, it has only to check the package name where the class is located and find the corresponding bundle very easy and vice versa.

Alternatives

As for the naming of the bundles, also for the packages other patterns exist, because our recommendation is not the only correct solution. But it's one where the naming of the packages and bundles are very consistent amongst each other. Note you have to consider the advantages and disadvantages of each naming pattern and if it's appropriate for your project.

But we don't want to go into details about the other package naming patterns, because it would break the size of this document.

8.3 Rich Client

Intention

This chapter contains instructions, recommendations and best practices for the development of a rich client application as a part of the reference architecture.

8.3.1 Extension Points

Overview

The Eclipse Extension Point concept enables a modular implementation of a whole application. The advantages are that, besides of a core implementation, additional bundles can be created for the extension of the application. This, without the need to change the implementation in the core.

In this chapter we give instructions and recommendations concerning the use of extension points based on our experiences during this thesis.

Recommendation

The Eclipse framework and the existing components provide a set of extension points for custom extensions. However, you can also create extension points for your own components.

We generally recommend using the same approach as Eclipse for the use of extension points. A core component provides various extension points for its extension and should be able to be used, independently from its extension components, on its own. By using appropriate visibility settings, the implementation of the core should be protected from changes. The intention is to have a core component which does not have to be changed. The changes are applied through extensions on the provided extension points.

To create a RCP application, we recommend using a base RCP package to start with. The already existing extension points can be used to create your application based on your requirements. If your application domain is larger, we recommend extending this base RCP package in order to create your own core implementation and the appropriate own extension points.

Concerning the creation of own extension points, we only recommend it in cases, where, from the sight of the base, extending functionality should be provided. Otherwise, in cases where functionality from the base should be used, we recommend using other mechanisms, such as aggregation, composition, inheritance and the like.

8.3.2 RCP Application

Overview

The implementation of an application using Eclipse RCP leads sooner or later to the discussion about its execution. Beside its integration as a further plug-in in Eclipse, it can also be executed as an independent application consisting of only the required components and plug-ins.

If the developed application is not an extension of Eclipse, it will be built and executed as an independent application. Eclipse RCP provides classes and interfaces to support the decision to implement an independent application. They allow a fast and compact extension of your RCP application with the required parts.

The following recommendations and alternatives are based on descriptions and comparisons listed in chapter 6.3.1. Additionally *[Vogel09]* and *[Burnette06]* provide good descriptions and tutorials to the recommendations.

8.3.2.1 Recommendation

Recommendat-

Eclipse RCP prescribes for building an own and independent RCP application

ion

the class structure shown in **Figure 39**. For some of them default implementation can be used depending on the decision either to do the configurations programmatically or through extension points.

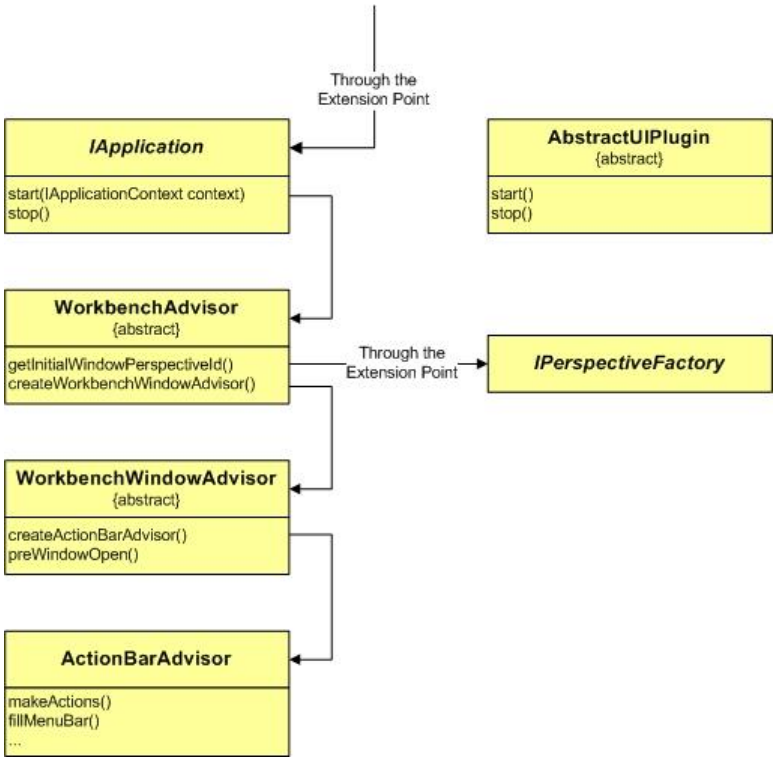

Figure 39: Required classes to build an independent RCP application

Own application project

For the development of an independent RCP application, we recommend you to locate the additional configurations and implementations in a new plug-in. The separation of this issue from the rest of the application has the advantage of more focused plug-ins with a minimum of dependencies, which is a general best practice. Additionally, an already existing plug-in doesn't have to be misapplied to provide these functions.

The additional configuration and management effort for the new plug-in is on the other hand marginal.

Use Extension Points

We also recommend you to do the necessary configurations of your application as much as possible with extension points. The reason for this recommendation is on the one hand a reduced implementation effort, because the default implementations for the `WorkbenchWindowAdvisor` and the `ActionBarAdvisor` can be used. On the other hand a configuration using extension points is more flexible and better to change and adjust than hard coded ones.

The first extension point you need is `org.eclipse.core.runtime.applications` to register the class implementing `IApplication`. It defines the application which should be initialized when the plug-in is started.

The second extension point is `org.eclipse.ui.perspectives`. It allows the definition of a perspective of the application, which should be shown after the startup. Beside a class also the visible name and the id must be declared, oth-

erwise the perspective cannot be configured as default in the method `getInitialWindowPerspectiveId` of the class extending `WorkbenchAdvisor`.

If action bars like a menu or a status line should be added to the application, then the extension point `org.eclipse.ui.menus` is required. It allows the definition of the whole action bars. For more details see chapter 8.3.5.

Application

Nonetheless some implementations are required. The code snippet in **Listing 140** shows an implementation of the `IApplication` interface with the `start` and `stop` methods, which initializes the workbench.

Please notice that you have to create the display in a first step to pass it afterwards in the `createAndRunWorkbench` method together with the workbench advisor class.

```
public class Application implements IApplication {

  public Object start(IApplicationContext context)
    // Create SWT Display
    Display display = PlatformUI.createDisplay();
    try {
      // Create the workbench advisor and run the Eclipse
      // workbench
      int returnCode = PlatformUI.createAndRunWorkbench(display,
                       new ApplicationWorkbenchAdvisor());

      if (returnCode == PlatformUI.RETURN_RESTART)
        return IApplication.EXIT_RESTART;
      else
        return IApplication.EXIT_OK;
    } finally {
      display.dispose();
    }
  }

  public void stop() {
    final IWorkbench workbench = PlatformUI.getWorkbench();
    if (workbench == null)
      return;

    final Display display = workbench.getDisplay();
    display.syncExec(new Runnable() {
      public void run() {
        if (!display.isDisposed())
          workbench.close();
        }
      });
  }
}
```

Listing 140: Example of the application class with the `start` and `stop` methods

In the `stop` method, the created workbench has to be closed to allow the system's disposal. Otherwise the workbench will not shut down correctly and cannot save the workbench settings.

Workbench

Another class which has to be implemented is the one extending the `WorkbenchAdvisor` class. The minimal implementation must override the method `getInitialWindowPerspectiveId`. It returns the id of the default perspective

which should be shown after the startup as a string.

Listing 141 shows the minimal implementation of a workbench advisor. Naturally further methods to implement some state-dependant sequences can be overwritten if required.

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor
{

  private static final String PERSPECTIVE_ID =
      "org.herasaf.tutorial.personlocator.ui.perspective";

  @Override
  public String getInitialWindowPerspectiveId() {
    return PERSPECTIVE_ID;
  }

  ...
}
```

Listing 141: Example of the workbench advisor class with the default perspective

In the example above the method `createWorkbenchWindowAdvisor` is not implemented. In this case Eclipse RCP uses a default advisor for the workbench window. Thereby the configuration of the workbench window cannot be made and those for the action bars only with extension points.

8.3.2.2 Alternatives

Alternatives

Alternatively to the recommended solution the structure and implementation can be varied within the given scope shown in **Figure 39**.

Integration in an existing plug-in

The first variation point is the plug-in. You can integrate the application classes and configurations in an existing plug-in. If this plug-in already acts as the main plug-in this is no problem, but to add the configurations and classes arbitrarily can increase dependencies unnecessarily and cause a bad design.

Therefore we can recommend this alternative only in cases where the plug-in acts already as the main plug-in.

Programmatic configuration

Another point you can vary from our recommendation is the configuration. You can do it in a programmatic manner instead of using extension points. In this case you have to invoke the class implementing `IApplication` in the activator and you must initialize the action bars in the class extending `ActionBarAdvisor`.

It is also a possible solution but has several disadvantages. The first one is the invocation of the application class in the activator. Normally, the activator should not invoke further application code if not necessary.

Another disadvantage is the programmatic configuration of the action bars. It is cumbersome to combine programmatic configurations with those made with extension points. Therefore, the action bars for the whole application should be implemented in the action bar advisor. A later extension in the plug-ins is not possible without visual side effects like wrong orders and further more.

Therefore we cannot recommend to do the configurations in a programmatically manner or only as a supplement to the declarative ones.

Workbench

In order to use an individual workbench window advisor instead of the default one provided by Eclipse RCP you must override the method `createWorkbenchWindowAdvisor` in the workbench advisor and return an instance of your own class. **Listing 142** below shows such a code snippet.

```
  @Override
  public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
      IWorkbenchWindowConfigurer configurer) {
    return new ApplicationWorkbenchWindowAdvisor(configurer);
  }
```

Listing 142: Example for the `createWorkbenchWindowAdvisor` method.

Workbench Window

The instance of the class you return in the workbench advisor must extend `WorkbenchWindowAdvisor`. It accompanies the creation process of the workbench window and allows its configuration. Therefore the state-dependant methods like `preWindowOpen` or some other must be overridden. **Listing 143** shows such a code snippet.

```
public class ApplicationWorkbenchWindowAdvisor extends Workbenchwindowadvisor {

  public ApplicationWorkbenchWindowAdvisor(
      IWorkbenchWindowConfigurer configurer) {
    super(configurer);
  }

  @Override
  public ActionBarAdvisor createActionBarAdvisor(
      IActionBarConfigurer configurer) {
    return new ApplicationActionBarAdvisor(configurer);
  }

  @Override
  public void preWindowOpen() {
    IWorkbenchWindowConfigurer configurer =
                                    getWindowConfigurer();
    configurer.setInitialSize(new Point(800, 600));
    configurer.setShowCoolBar(true);
    configurer.setShowFastViewBars(false);
    configurer.setShowPerspectiveBar(true);
    configurer.setShowProgressIndicator(false);
    configurer.setShowMenuBar(true);
    configurer.setShowStatusLine(true);
    configurer.setTitle("Person Locator");
  }
}
```

Listing 143: Example of the workbench window advisor class with the window settings

Additionally, you can override the method `createActionBarAdvisor` to return an instance of your own class. It's optional and must therefore not be overridden, only when you want to implement it on your own.

Action Bar

Such a class must extend `ActionBarAdvisor` and accompanies the creation and configuration of the action bars. To add actions to one of the action bars you must register them in a first step in the `makeActions` method. Afterwards they can be added in the corresponding fill method to the given action bar. The **Listing 144** below shows such a programmatic configuration of the menu bar.

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor
{

  public ApplicationActionBarAdvisor(IActionBarConfigurer confi-
gurer) {
    super(configurer);
  }

  @Override
  protected void makeActions(IWorkbenchWindow window) {
    super.makeActions(window);

    register(ActionFactory.QUIT.create(window));
    register(ActionFactory.UNDO.create(window));
    register(ActionFactory.REDO.create(window));
    ...
  }

  @Override
  protected void fillMenuBar(IMenuManager menuBar) {
    super.fillMenuBar(menuBar);

    MenuManager fileMenu = new MenuManager("&File",
                        IWorkbenchActionConstants.M_FILE);
    menuBar.add(fileMenu);

    fileMenu.add(new Separator(
                        IWorkbenchActionConstants.FILE_START));
    fileMenu.add(getAction(ActionFactory.QUIT.getId()));
    fileMenu.add(new Separator(
                        IWorkbenchActionConstants.FILE_END));

    ...
  }
}
```

Listing 144: Example of the action bar advisor class

But note, the combination of programmatic composed action bars with extension points defined in other plug-ins can lead to problems. Some of them are duplicate action bar entries, wrong order of the entries and further problems. Therefore consider such an implementation in detail before you decide to use it.

8.3.2.3 Hints

Generation of an application skeleton

You can generate the explained skeleton for a RCP application using the creation wizard. During the creation of a new plug-in project, an option with the question “Would you like to create a rich client application?” exists. When you choose the option *yes*, the whole skeleton will be generated. You only have to adapt it.

8.3.3 RCP Plug-in

Overview

Different to the implementation of an independent RCP application, the development of RCP plug-ins is very easy. Eclipse RCP has no prescriptions for its implementation expect for the existence of the *plugin.xml* and *Manifest* files.

But for an easier and faster development Eclipse RCP provides some basic classes you can use. In this chapter we give you recommendations for them based on descriptions and comparisons listed in chapter 6.3.3.

Recommendation

In some cases it is needed to implement an activator class in your plug-in to provide utilities for the whole plug-in or other purposes. We recommend you to use for its implementation the class `AbstractUIPlugin` provided by Eclipse RCP. It provides a wide range of very useful utilities instead the implementation of the `BundleActivator` interface.

Alternatives

Naturally you can also use the `BundleActivator` interface but you must handle common parts like utilities by your own or must implement them first. Only in cases where you don't need them, we recommend the implementation with the interface.

8.3.4 Perspectives & Views

Overview

Eclipse RCP provides the possibility to define views and perspectives. Whereas the views are use to group the UI elements into logical parts, are the perspectives used to compose whole user interfaces on the basis of the views.

Such functionality gives not only the developer the possibility to define default layouts, but also the user to edit it individually. For that reason this is an important feature.

8.3.4.1 Perspectives

Definition

For the definition of perspectives Eclipse RCP provides generally only the possibility using the extension point `org.eclipse.ui.perspectives`. Otherwise it would be necessary to add them programmatically to the application setup classes. But this is not possible, if you extend an existing application for example. That's why the declarative way is the only solution. Additionally, it corresponds with our recommendation to replace programmatic configurations through declarative ones.

The definition itself is very simple. Beside an id and a name you only have to create a class implementing the `IPerspectiveFactory` interface, like the code snippet in **Listing 145** below shows.

```
<extension point="org.eclipse.ui.perspectives">
  <perspective
    class="org.herasaf.tutorial.ui.Perspective"
    id="org.herasaf.tutorial.ui.perspective"
    name="Default">
  </perspective>
```

```
</extension>
```

Listing 145: Example a perspective definition using the extension point `org.eclipse.ui.perspectives`

Composition

But this is only for the definition of the perspective, it contains nothing yet. So it's required to add views to it. Eclipse RCP provides two ways to achieve this, a programmatic and a declarative one.

As for the perspective definition we recommend you to use the declarative way using the for extension point `org.eclipse.ui.perspectiveExtensions` its composition. It's the more flexible solution compared with the programmatic way, especially for later extensions.

In the extension point you have to specify a `perspectiveExtension` with the definition which perspective you want to extend. Here you can add views as well as perspective, view and further shortcuts for the menus. **Listing 146** shows such a composition using the extension point.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.ui.perspective">
    <view closeable="true"
      id="org.herasaf.tutorial.ui.Navigator"
      minimized="false"
      moveable="true"
      ratio="0.25"
      relationship="left"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      standalone="true"
      visible="true">
    </view>
    <perspectiveShortcut
      id="org.herasaf.tutorial.ui.perspective">
    </perspectiveShortcut>
    <viewShortcut
      id="org.herasaf.tutorial.ui.Navigator">
    </viewShortcut>
  </perspectiveExtension>
</extension>
```

Listing 146: Example a perspective composition using the extension point `org.eclipse.ui.perspectiveExtensions`

Alternative composition

The alternative way is to compose the perspective in the class you had to create for its definition. Although you have in a typical case access to these classes, you have to do code changes, which should be avoided. Additionally, such a solution is less flexible compared with the declarative way.

In the method `createInitialLayout`, which must be implemented through the inheritance of the `IPerspectiveFactory`, you can add the views to the passed `IPageLayout` object. Also the addition of perspective shortcuts is possible, like shown in **Listing 147** below.

```
public class Perspective implements IPerspectiveFactory {
```

```
  public static final String ID =
      "org.herasaf.tutorial.ui.perspective";

  public void createInitialLayout(IPageLayout layout) {
    layout.addView("org.herasaf.tutorial.ui.Navigator",
      IPageLayout.LEFT, IPageLayout.DEFAULT_VIEW_RATIO,
      IPageLayout.ID_EDITOR_AREA);
    layout.addPerspectiveShortcut(ID);
    layout.setEditorAreaVisible(true);
  }
}
```

Listing 147: Example a perspective composition using the perspective class

8.3.4.2 Views

Definition

Very similar to the definition of the perspectives are those for the views, but with a different extension point `org.eclipse.ui.views`. Beside the id and the name you have to create a class implementing ViewPart. Additionally, you can define a category, which is used for grouping purposes in the view switch dialog.

Listing 148 below shows a code snippet with such a definition.

```
<extension point="org.eclipse.ui.views">
  <view
    category="org.herasaf.tutorial.ui.category"
    class="org.eclipse.ui.navigator.PersonNavigator"
    id="org.herasaf.tutorial.ui.Navigator"
    name="Navigator"
    restorable="true">
  </view>
  <category
    id="org.herasaf.tutorial.ui.category"
    name="Person Locator">
  </category>
</extension>
```

Listing 148: Example a view definition using the extension point `org.eclipse.ui.views`

Alternative solutions for the definition using the extension point do not exist, because for their assignment the id is required, which must be known by RCP. Otherwise it would be necessary to publish the id programmatically, which is cumbersome and more error-prone.

Implementation

For the implementation you must inherit your view class from `ViewPart` or a derivation of it. Through overriding the method `createPartControl`, which is mandatory, you can build up your user interface using UI elements like buttons, fields and further more.

Listing 149 shows a shortened example of such a view implementation using the `ViewPart` class.

```
public class PersonNavigator extends ViewPart {
  ...

  public PersonNavigator() {
```

```
    super();
  }

  @Override
  public void createPartControl(Composite parent) {

  this.parent = parent;

  // Composition of the view with UI elements
  ...
  }

  public void setFocus() {
  }
}
```

Listing 149: Example a view implementation using the abstract class `ViewPart` as base

8.3.4.3 Hints

Getting the shell

In the views it can be a problem getting the corresponding Shell instance, which is necessary to show dialogs and further purposes. Through the getAdapter method of the extension interface it may be available, but not for sure.

You can get it in every case through the statement `getSite().getShell()`. This allows you also to extend your `getAdapter` method to return the `Shell` if it doesn't yet.

8.3.5 Menu- & Toolbar

Overview

The menu and the toolbars are a very important for the usability of an application. They are connected with different views and parts in the application and provide global actions and commands. In this chapter, we are going to give a recommendation based on our research and the gained experiences during this thesis.

Definition of commands

The menus of the application can be extended with additional functionality by using commands. The commands have to be specified using the extension point `org.eclipse org.eclipse.ui.commands`.

The extension point gives the possibility to create categories for commands. We recommend using this feature and categorizing the commands based on their main concerns. After the definition of a command category, it can be assigned in the `categoryId` field of the command. This feature helps keeping an outline over the commands.

The extension point also allows specifying a default handler for a command. This is done through the attribute `defaultHandler`. The default handler has to implement the interface `IHandler` and is active if no other handler is active. Please note that this handler will conflict with handlers that do not specify `activeWhen` conditions. Further details about action handling are available later in this chapter.

The command framework also provides an extension point called `org.eclipse.ui.commandImages`. It enables to link different types of images and icons (`image`) to a particular command. These images can be used

wherever the command appears in the user interface. We recommend using this feature, because it avoids the necessity to adjust the configuration in multiple places in the case of icon changes.

Positioning commands

The extension point `org.eclipse.ui.menus` allows adding various custom additions to the menus in the framework.

The positioning of the commands is done by adding a new `menuContribution` and specifying a `locationURI` for it. The `menuContribution` itself contains references to existing commands. As for the use of optional attributes we recommend defining an icon (`icon`), specifying a label (`label`) and defining a tooltip (`tooltip`) for these command references. Additional features like `style`, `disabledIcon`, `hoverIcon`, and others should be used depending on the requirements of your application.

The `locationURI` string specifies the command location (main menu, context menu, toolbar …). The following pattern describes how the insertion point is defined:

```
<scheme>:<menu-id>[?<placement-modifier>]
```

Listing 150: Pattern for the definition of the menu contribution's `locationURI`

The scheme can have the values `menu`, `toolbar` or `popup`. The value `menu` adds a contribution to the main menu or to the menu of the view. In this case the `menu-id` has to point to an existing view or to the standard Eclipse menu with the id `org.eclipse.ui.main.menu`. The `placement-modifier` is specified based on the pattern `<placement>=<id>`. For `placement`, the tags `before` or `after` can be used, whereas the `id` can be an existing separator name, menu id or item id.

The scheme `toolbar` contributes to any toolbar of the application. The `menu-id` can point to any existing view or to the `org.eclipse.ui.main.toolbar` Eclipse main toolbar. The `placement-modifier` can be used here, too.

With the scheme `popup` contributions to the context menu can be realized. The `menu-id` can point to any registered context id or to `org.eclipse.ui.popup.any` to contribute to all registered context menus. The `placement-modifier` can also be used.

The class `MenuUtil` contains some common constant values for menu URIs, which can be quite helpful during the implementation.

Command handling

The behavior for additional functionality and the handling of the commands in the text editor can be done in two ways.

The first way is defining a handler through the extension point `org.eclipse.ui.handlers`. A handler has to implement the interface `IHandler` and is linked to a command through the attribute `commandId`. With the `activeWhen` clause the activation of the handler can be configured. The `enabledWhen` clause allows configuring the circumstances in which the handler is visible.

As for the concrete implementation of the handler, the active workbench window can be acquired with the received `ExecutionEvent` using the `HandlerUtil.getActiveWorkbenchWindow` static method. Through the workbench window further information, like e.g. the selection (`ISelection`) in the text edi-

tor, can be retrieved.

The other way is defining an `editorContribution` for the text editor in the extension point `org.eclipse.ui.editorActions`. This is done by specifying the attribute `targetId`, which has to point to a valid text editor definition. The `editorContribution` contains actions, which can be linked to commands through the use of the attribute `definitionId`. The implementation class (`class`) of the action has to implement the interface `IEditorActionDelegate` and the `definitionId` attribute has to point to a valid command. As for the attribution we recommend specifying a tooltip (`tooltip`) besides the required label (`label`). Additional features like `style`, `disabledIcon`, `hoverIcon`, and others should be used depending on the requirements of your application.

As for the implementation, the `run` method in the action implementation is invoked each time the command is executed. Additionally, the selection (`ISelection`) is given through the method `selectionChanged` every time the user changes the selection in the text. The `setActiveEditor` method is invoked each time the editor is activated and gives access to the new editor target and thus the possibility to handle editor changes.

Because of the proper specification and design of the command framework, which gives a lot of advantages (see comparison in chapter 6.3.5.3.1), we generally recommend using command handlers. In some cases, where you should be able to react on selection changes in the text and/or handle editor changes and activations, the use of editor actions can be reasonable.

Dynamic actions

There are cases where a dynamic creation of actions should be possible. During the implementation we considered the creation of dynamic actions, although we did not use it in our solution in the scope of this thesis.

In order to be able to create dynamic entries based on the application lifecycle, a programmatic approach has to be chosen. The `EditorActionBarContributor` class of the Eclipse framework provides different methods for the contribution of actions. These methods provide access to the `IMenuManager` and thus make it possible to add custom actions (`IAction`) into it. Like this, it is possible to create dynamic actions based on the current application state.

Depending on your requirements, an already existing `IAction` can be used or a custom implementation has to be provided. It is important to set the action an id through the method `setId` in order to be able to identify the action when it is added to a contribution manager.

Another issue with the creation of dynamic entries is the refresh of the context menu in the case of changes. Having the implementation in the text editor contributor class, the text editor view has to be reopened in order to refresh the context menu entries. In the scope of this thesis, we did not analyze how a refresh can be realized without the need to reopen the editor.

8.3.6 UI elements

Overview

In this chapter we discuss design approaches for UI elements. How can you apply them and which is the correct way. We compare the several solutions of each approach and give you recommendations.

8.3.6.1 UI element implementation style

Overview

The problem to reduce the amount of duplicate code is especially in the UI programming a huge challenge. Nothing else uses as much inner classes, temporary variable and duplicated code as UI classes. To allow an easy maintenance and extension at any time, the reduction of duplicate code is a main issue.

To allow a fast and clean development of user interfaces as well as its extension and maintenance, a minimal and stable implementation should be possible.

The following recommendations are based on the comparisons and decisions described in chapter 6.3.6.1.

Recommendation

For your development of the user interface, we recommend you to use the raw UI elements in the cases where you have no or only a small amount of duplicate code sequences to configure them. Or in cases you don't have to extend the existing functionality of the elements. In the most cases it is the most appropriate solution, because the implementation of subclasses or wrappers would result only in forwarding object, with no additional logic.

For the implementation of own UI elements which combine or extend the functionality of existing elements, we recommend you to use the wrapper implementation instead the subclassing. The reason for this recommendation is the implementation of them with fewer troubles, because you don't have to consider not using protected or private states to achieve an easy portability between the different platforms (Windows, Linux, etc.). Additionally you don't have to override the `checkSubclass` method to prevent the subclass check.

It is also the recommended implementation style of the Eclipse SWT developers *[SWTSubclass]* to extend UI elements of exactly the reasons mentioned above.

But all in all you have the same trade-off like in the Adapter Pattern described by the Gang of Four *[GoF1995]*, even though it is more the Decorator Pattern in our case. Should you subclass the existing class and override only the methods you want to extend or should you implement a wrapper object with passes some of the method calls directly to the wrapped object?

Alternatives

Therefore the subclassing can be a real alternative in cases, where the wrapper would be only an object forwarding the method calls to the wrapped object.

We don't recommend it explicitly because it may be an appropriate solution if you consider the restrictions mentioned in chapter 6.3.6.1. But you should avoid using this solution whenever you can.

8.3.7 User Layout

Overview

By default, the Eclipse framework does not save the user interface layout when the application is closed. However, the framework provides a functionality, which is already implemented and simply has to be activated in order to come into operation.

Recommendation

We generally recommend using the save and restore feature provided by the Eclipse framework. It really makes the use of the application more comfortable and saves a lot of time by avoiding the need to customize the user interface on every startup.

The implementation is quite simple. All you need to do is implementing the `initialize` method in your `WorkbenchAdvisor` class of your UI plugin and enabling the save and restore feature on the `IWorkbenchConfigurer` through the use of the `setSaveAndRestore` method.

8.3.8 Graphical Editor

Overview

The graphical editor is undisputed a powerful tool for any kind of visualization, like diagrams, which should be editable. Eclipse provides with the *Graphical Editing Framework (GEF)* and the *Graphical Modeling Framework (GMF)* two projects treating with this challenge.

The basis provided by the two projects allows the development of an easy and powerful graphical editor in an uniform manner and a minimum of time. Through their comprehensive approach, a wide range of different interactions and further features are easy to configure and implement.

In this chapter we show you how you can create and configure your own graphical editor. For its different components, we give you advices and recommendations based on our experiences gained during the implementations.

Encapsulation

As a first advice for the implementation of your graphical editor, we recommend you to encapsulate the different components in a useful manner. This means for example the definition of general interfaces for the implementation of the components. You can benefit from synergies through the general implementations which can be reused several times.

We also recommend you to isolate the common parts and implementations strictly from the specific ones for the same reason. Otherwise you have to implement every time the whole bunch of classes newly or copy it from an existing application.

8.3.8.1 Technologies

Intention

For the implementation of a graphical editor the *Graphical Editing Framework (GEF)* as well as the *Graphical Modeling Framework (GMF)* exist, whereas the GMF is more a combination of GEF and EMF as a totally new framework.

The choice for the right framework depends mostly on the context in which you are going to use the graphical editor. Both of them have individual advantages and disadvantages and are suited for different situations.

GEF

Based on the comparisons of the frameworks in chapter 6.3.8.1, we recommend you to use GEF in cases where you are going to integrate the graphical editor extensively in the remaining parts of your application. This for example to display and manipulate your domain model or parts of it graphically or if the main purpose of your application is the graphical editor.

Also, when you need special behavior and restrictions in your graphical editor

GEF is the appropriate choice, because you can define it in detail on your own. So you have a wide range of possibilities to vary your implementation and to adjust it on your requirements.

GMF

Very different is GMF. We recommend you to use it only in cases where you need a graphical editor, which is widely isolated from the remaining parts of your application. It is also very useful if you don't want to adjust it with specific behavior or restrictions, because its adaptability is limited through the configuration models and the code generation.

Additionally, GMF brings a whole bunch of disadvantages which should be well considered. Otherwise you will invest too much time for its configuration and adaptation.

Because of our decision in chapter 6.3.8.1 and our requirement for extensively integrated graphical editors, we will describe in the further chapters only GEF-centric descriptions and recommendations.

For further information about GMF, see *[GMF]*, which provides detailed descriptions and links to several tutorials.

8.3.8.1.1 Hints

Which technology is the appropriate one?

We cannot tell you which technology is the appropriate one for your solution. It depends strongly on your requirements and the purpose of the further application area. But generally we can say GEF is the more appropriate one if you intend an extensive integration in the remaining parts of your application, whereas GMF is very useful for standalone editors. Additionally, GEF allows a strong customization which is not reachable with the GMF configurations.

8.3.8.2 The Editor

Intention

The editor class is the main part of the graphical editor, because it acts as its root, whereas all other parts are either directly or indirectly dependent from it. For the most of them he's responsible to instantiate or at least to provide them through the specific methods.

Schematic class diagram

The **Figure 40** shows the different parts of the graphical editor in a schematic class diagram.

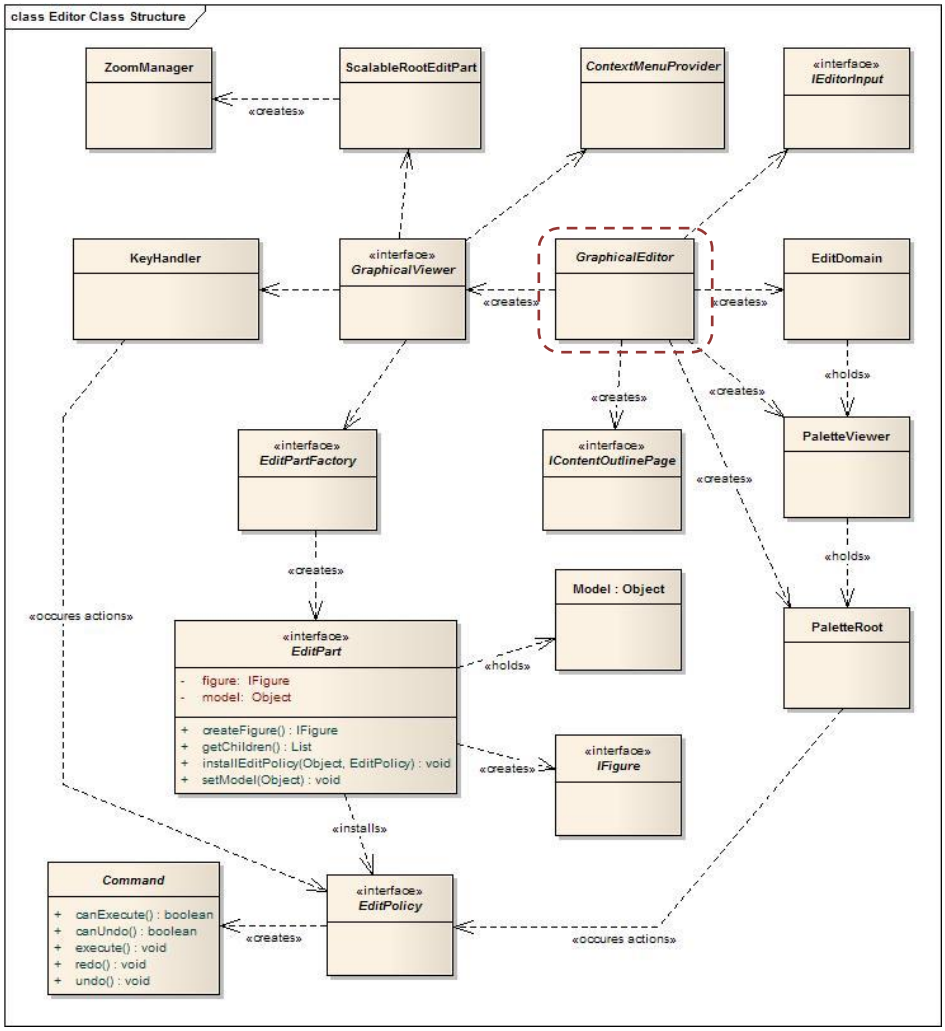

Figure 40: Schematic diagram of the graphical editor (red) with the classes and interfaces of the dependent parts

Implementation

For the implementation of the graphical editor, it's recommended to start with the editor class itself, otherwise the dependent parts cannot be tested respectively the graphical editor cannot be executed.

GEF supports its implementation with the abstract classes `GraphicalEditor`, `GraphicalEditorWithPalette` and `GraphicalEditorWithFlyoutPalette`, whereas the second and third ones are derivations of `GraphicalEditor`. All of them provide the fundament of a graphical editor but with a different maturity level and consequently different base functionality.

We recommend you to use either `GraphicalEditorWithPalette` or `GraphicalEditorWithFlyoutPalette`, because they provide additional functionality compared with their parent class. Several methods required for the implementation of the further features are already fully provided or at least prepared for its usage. Therefore we recommend the use of the `GraphicalEditor` class only in cases where the additional functionality won't be used.

Common parts

Additionally, we recommend you to split the implementation of the editor in two classes, an abstract super class and a concrete derivation of it. The super class

contains all general sequences which apply to all of your editors, whereas the derived classes contain the sequences for a specific editor. The reason for this advice is mainly the use of synergies through the reusability of the common parts. Otherwise you have to implement them again for each editor, which leads to duplicate code.

Configuration & Initialization

For the implementation and configuration of the editor classes, you can override different methods. Most of them are hook methods and will be invoked in a given lifecycle, which allows you wide configuration possibilities. Please note, you should invoke in these methods also the sibling method in the super classes, otherwise some of the provided functionality will not be available anymore.

Very important methods for the configuration and initialization of the graphical editor are the following ones.

- `init` is one of the first methods invoked after the instantiation of the editor class. The already existing implementations in the super classes set the passed site and input. We recommend you to only do configurations, which cannot be made in the constructor, in this method. An example for such a situation is the wiring with the Spring Framework. In this case you should preferably set the `EditDomain` in this method. Otherwise you may get exceptions.
- `configureGraphicalViewer` is invoked after the `init` method. Its purpose is to configure the graphical viewer before it receives its contents. The GEF developers propose to instantiate the `EditPartFactory` in this method.
  We recommend you to configure in this method the `EditPartFactory` as well as menus and handlers, like the context menu, the key handlers or the zoom manager.
- The last method for initialization purposes is `initializeGraphicalViewer`. The Javadoc suggests for this method to set the content of the graphical editor. We recommend you to do exactly this. Maybe for registering listeners this method is also appropriate. But we disadvise for doing further configurations here.

In the code snippet shown in **Listing 151** we give you an example for such an implementation. The creation of the context menu, the zoom manager and the key handlers in this example are outsourced in own factories. We recommend you to do it the same way, because it allows an easy exchange of different implementations.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  // EditPartFactory
  viewer.setEditPartFactory(editPartFactory);

  // Context Menu
  viewer.setContextMenu(contextMenuProviderFactory
      .createContextMenuProvider(viewer, getActionRegistry()));

  // ZoomManager
```

```
  viewer.setRootEditPart(zoomManagerFactory
      .createZoomManager(this));

  // KeyHandler
  viewer.setKeyHandler(keyHandlerFactory
      .createKeyHandler(this));
}

@Override
public void init(IEditorSite site, IEditorInput input)
    throws PartInitException {

  setEditDomain(new DefaultEditDomain(this));
  super.init(site, input);
}

@Override
protected void initializeGraphicalViewer() {
  GraphicalViewer viewer = getGraphicalViewer();
  viewer.setContents(contents);
}
```

Listing 151: Examples for the editor's configuration and initialization methods

Hook methods

The editor class enables you to hook-in special configurations and implementations through additional methods. Most of them are invoked through GEF in the editor's lifecycle and have different purposes.

Through `commandStackChanged` you will be notified about changes in the command stack, which is used for the execution of the commands and the corresponding undo/redo functionality. Here, you have the possibility to mark the editor as dirty, which results in a visual representation of it in the tab.

A little different are `doSave`, `getPaletteRoot` and `setInput`, which allow intervening in the lifecycle, for example to save the editor's content or to set its content through the passed input. In the case of the `dispose` method, you can clean up resources before the editor will be destroyed.

We recommend you to override these methods if you require them for your graphical editor implementation. But consider invoking the methods of the parent class, otherwise you will lose some of the already provided functionality. The code snippet in **Listing 152** shows an example of such an implementation overriding the setInput method.

```
@Override
protected void setInput(IEditorInput input) {
  super.setInput(input);

  if (input instanceof IExtEditorInput) {
    contents = ((IExtEditorInput) input).getEditorInput();
  }
}
```

Listing 152: Example implementation of an editor's hook method (`setInput`)

Adaptability

Very different is the `getAdapter` method, which is part of Eclipse Extension Interface Pattern *[ExtensionInterface06]*. It allows the extension of the graphical editor with further functionalities, but without the force to provide them through

implementing interfaces and methods.

The method must be used to provide the outline corresponding to the graphical editor. Therefore we recommend you to use this method for further extensions, because you can simply invoke this method to get them, instead of casting the editor class to access the additional provided functions.

8.3.8.2.1 Hints

Wiring with Spring

As for the wiring with Spring, we recommend you to prepare your graphical editor class to allow an easy wiring. For this purpose, the different components you intend to inject must be class global attributes and need a corresponding setter method. Otherwise, they cannot be injected properly and you editor will fail during the opening.

To allow the wiring of the editor with Spring in a RCP application you must use the Spring Framework Utilities *[Lippert08]*, which allow the injection of beans directly in the extension point. For this purpose you must replace the editor class through `org.eclipse.springframework.util.SpringExtensionFactory` followed by a colon and the name of the bean in the application context file. **Listing 153** below shows such an injection of a graphical editor.

```
<extension point="org.eclipse.ui.editors">
  <editor
     class="org.eclipse.springframework.util.
            SpringExtensionFactory:graphEditor"
     contributorClass="org.herasaf.tutorial.personlocator.
            ui.graph.PersonLocatorActionBarContributor"
     default="true"
     id="org.herasaf.tutorial.personlocator.ui.graph.
            PersonLocatorGraphEditor"
     name="Statement Editor">
  </editor>
</extension>
```

Listing 153: Editor injection with Spring in the extension point `org.eclipse.ui.editors`

Show the dirty state

Sometimes the visibility of the dirty state of the graphical editor is not shown correctly. It seems to be a problem in the base class of the graphical editor. But you can fix it very fast through overriding the method `commandStackChanged`. Beside the invocation of the method in the super class, you must fire a property change event, as shown in **Listing 154**.

```
@Override
public void commandStackChanged(EventObject event) {
  super.commandStackChanged(event);

  firePropertyChange(PROP_DIRTY);
}
```

Listing 154: Example implementation of an editor's hook method (`setInput`)

Through the notification of the graphical editor about the dirty state, it is now

able to visualize the current state correctly.

Prevent EditDomain problems

If you wire your graphical editor with Spring and try to use it, you may get a `NullPointerException`, especially if you try to open the editor more than once. The cause of this problem is mainly the initialization of the `EditDomain`.

You can prevent this problem in two ways. The first is to check if the editor contains already an `EditDomain` object. Only in cases where it is missing you have to create a new one.

The second and in our opinion the better solution is to set the scope of the editor bean in your application context file to prototype (`scope="prototype"`). For each call of the Spring Framework's `getBean` method a new instance will be created. Additionally, Martin Lippert *[Lippert08]* also suggests this way in his blog.

8.3.8.3 The Editor Input

Intention

The editor input is an important component of the graphical editor. Its main purpose is to provide the editor with the required content. Using the editor without editor input is not possible, because it manages the editor instantiation.

But the editor input is not a container or model which holds the content; it's more a lightweight descriptor of it. The reason for such a design decision lies in the navigation history of the Eclipse workbench. It tends to hold the editor inputs for the reconstructions at a later time. Therefore it can hold quite a few of them. If the editor input would contain the content, this would lead to huge memory consumption.

Integration

For the integration of the graphical editor in an application with further components, where all of them use the same common data, like a domain model, we recommend you to use the editor input. The advantage of such a solution is the independence of the graphical editor from the way how the common data is managed and hold. Additionally, you can implement transformations of the common data in a style required for the graphical editor transparently.

The alternative solution would be using the `NullEditorInput` and to implement the graphical editor in a way that gets and sets the data he represents directly. A result of such a solution would be direct dependencies to the common data. Additionally, a transformation in a style required for the editor would be directly included in the graphical editor.

Implementation

Eclipse provides for its implementation the interface `IEditorInput` and some general implementations for it. They are very generic and can be used as a basis for nearly every editor. For developing own ones, this interface or an implementation of it must be used.

We recommend you to create in a first step an extended interface, which contains additional methods for getting and setting the content. This allows the implementation of the graphical editor against this interface which results in a very generic solution and a higher reusability. **Listing 155** shows an example of such an interface.

```
public interface IExtEditorInput extends IEditorInput {
```

```
  public Object getEditorInput();

  public void save(Object editorData);
}
```

Listing 155: Example of an extended `IEditorInput` interface

Based on this interface you can implement your editor input, using a parser, accessing a file or something else. But notice that you should implement the `equals` and `hashCode` methods, because a new instance will be compared against all open editors. If one of them results in a match, no new editor will be opened.

We also recommend you to return an icon in the method `getImageDescriptor`. Otherwise you risk that the workbench will ignore your editor. If you don't have an icon or image, you can use the statement `ImageDescriptor.getMissingImageDescriptor()` instead, which provides an empty image.

8.3.8.4 Diagram Elements

Intention

The diagram elements are the component of the graphical editor which represent and visualize the real content. Without them nothing would be displayed in the editor. Therefore this is the core component.

Compared with the other components, this one covers several classes and interfaces and has therefore a larger size. It is organized in a MVC-like *[MVC07]* structure with a model, a view and a control layer, all with their own responsibilities.

8.3.8.4.1 Structure

Introduction

The diagram elements are organized in a MVC structure as described in the intention of this chapter, whereas the controller has the lead. It manages the model and creates the corresponding view, also called figure.

The controller, also known as edit part, acts as the mediator between the model, which holds the data, that should be visualized, and the figure, that represents the visual part. It is responsible to handle the requests on the figures respectively to forward them, as well as synchronizing the model and the view.

Schematic class diagram

The following diagram in **Figure 41** shows the dependencies between the different parts of the diagram elements.

The *view* is only responsible to visualize the information out of the model using labels, borders, various shapes and further more.

The opposite is the *model*, which holds the information about what should be visualized. But it does not care about the visualizations in detail. It only provides a notification mechanism to notify the dependent controllers and views.

And in-between is the *controller* which must synchronize these two elements. For this purpose it registers edit policies which handles the user interactions and update the figures after receiving notifications. Additionally, it's responsible to

provide GEF with the required information for the creation of the whole diagram, like sub elements of the represented model object. This is the reason, why for each model object a corresponding edit part and figure exists. Aberrations from this default are possible but not recommended.

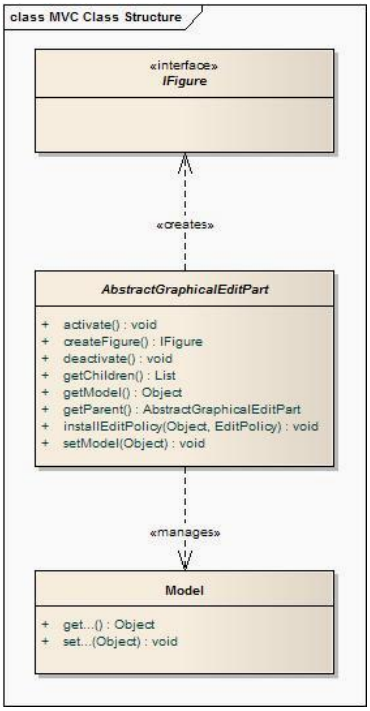

Figure 41: Structure of the graphical editor's diagram elements

Factory

As a completion to the MVC-structure additionally a factory exists, whose purpose is to instantiate new edit parts for a given model object. Otherwise, the edit parts itself has to manage the creation the further edit parts, which leads to a bad design.

This structure is given and limits the possibilities for own design decisions. But this is required, otherwise the structure would be too open and an efficient handling could not be reached or only with limitations.

8.3.8.4.2 Model

Introduction

As mentioned above, the model classes hold the information, which should be visualized in an individual way. For this purpose, the model must provide the data through accessor methods to be able to get and set them.

Different to the common domain model, the one for the graphical editor must be prepared for it. This means, it must additionally consist of classes which represent connections, if they should be visible. Otherwise, their handling would be very difficult which leads to fragile code. We recommend preventing such solutions.

Figure 42 below shows the difference between a conventional domain model

and one used for the graphical editor. Beside an additional class *Diagram* also ones for the visualization of the connections between *Person* and *Address* and *Person* and *Location* are required. Further, the diagram must hold all classes which are displayed as nodes. Otherwise, no new elements can be created because the connections between them are created in a later step.

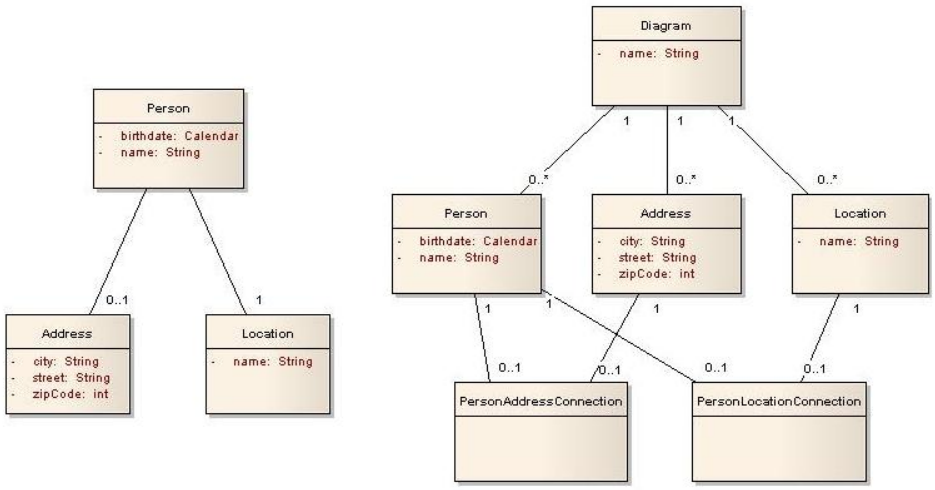

Figure 42: Conventional domain model (left) and the corresponding diagram model (right)

If the *Address* and *Location* are located in the *Person* node, the connection only for the additional class *Diagram* is required. Additionally, the dependencies between the classes are the same, because no connections are intended to be visualized.

Common parts

We also recommend you to outsource the common parts in super classes. This allows using synergies between the different classes through the reduction of duplicate code. Possible data for this purpose are the children elements or source and target connections, as shown in **Listing 156**.

```
public abstract class Node implements PropertyChangeEvents {

  protected String name = "<<...>>";
  protected Rectangle layout = new Rectangle();
  protected List<Node> children = new ArrayList<Node>();
  protected List<Connection> sourceConnections =
      new ArrayList<Connection>();
  protected List<Connection> targetConnections =
      new ArrayList<Connection>();

  protected PropertyChangeSupport listeners;

  ...
}
```

Listing 156: Excerpt of the model super class containing common parts for derived classes

It is also possible to implement your models without the outsourcing of the common parts. But this leads inevitably to more and duplicated code which increases the risk of errors. So we disadvice implementing such a solution.

Specific parts

Complementary are the classes representing the specific parts of the model. We recommend inheriting them from the appropriate class of the common part and extending them with the additional attributes and methods. How you implement them is your choice, because GEF makes no prescription.

Notification

Additionally, it's required to extend your model classes with a notification mechanism. Based on the small comparison in chapter 6.3.8.4.1.1 we propose the usage of the *JavaBeans property change* mechanism. In our opinion, it is the most appropriate solution for the model classes, because they are typical Java Beans. So this mechanism is developed in specific for exactly this purpose.

Certainly you can use other notification algorithms like Java's Observer implementation or own solutions. Normally they are designed in a more common way, not specific for Java beans. This may be a disadvantage.

8.3.8.4.3 View

Introduction

On the contrary side of the model is the view. It doesn't manage the data; it only cares about its visualization. Therefore, GEF provides for this purpose the interface `IFigure` and numerous default implementations of it, like rectangles, circles, labels, buttons and further more.

The implementation of own figures respectively the adaption of existing ones is very similar to Java Swing *[SUNDesktop].* You also have to add the elements which should be displayed in your figure. For their organization you can set a `LayoutManager`.

Implementation

We recommend you to use the default figures, if these satisfy your needs. This especially in case where you don't want to adjust them. For example if the only contained elements are the figures of the children model, which will be added to it automatically.

Own figures

Only in cases where your requirements cannot be satisfied with the default figures, we suggest you to implement own figures. Although the implementation is simple and the effort low, you should consider a good design for them.

For this reason we recommend you to define in a first step an interface for your figure inheriting from `IFigure`. It allows you to exchange the concrete implementation of the figure easily without huge refactoring efforts. Additionally, it's possible to outsource the instantiation of the figure in an own factory, without knowing what exact figure will be created. **Listing 157** shows such an interface.

```
public interface INodeFigure extends IFigure {

  public void setLayout(Rectangle layout);

  public void setBackgroundColor(Color background);

  public void setForegroundColor(Color foreground);

  public String getText();

  public void setText(String text);
```

```
  public Rectangle getTextBounds();
}
```

Listing 157: Example of an interface defining an own figure

In a later step you can implement your figure using the defined interface. We recommend you to inherit from an already existing implementation of the `IFigure` interface, like a rectangle, circle or something else. GEF provides you with numerous default implementations for nearly each situation. Otherwise, you must implement all the methods defined in the `IFigure` interface on your own.

For the arrangement of the elements you want to display in your figure, you can use one of the layout managers provided by GEF. All of them implement the `LayoutManager` interface directly or indirectly.

The **Listing 158** shows an example of such a figure implementation using the interface defined above. It uses additionally the `RoundedRectangle` as a default implementation of a figure as its base. For the organization of the figures the `ToolbarLayout` is used, which arranges the elements in a single row or column.

```
public class NodeFigure extends RoundedRectangle implements INodeFigure {

  private Label nameLabel = new Label();
  private ToolbarLayout layout = new ToolbarLayout();

  public NodeFigure() {
    layout.setMinorAlignment(ToolbarLayout.ALIGN_CENTER);
    setLayoutManager(layout);
    nameLabel.setFont(new FontRegistry().getBold("Verdana"));
    add(nameLabel);
  }

  public void setBackgroundColor(Color background) {
    super.setBackgroundColor(background != null ? background
        : ColorConstants.white);
  }

  public void setForegroundColor(Color foreground) {
    nameLabel.setForegroundColor(foreground != null ? foreground
        : ColorConstants.black);
  }

  public void setLayout(Rectangle layout) {
    getParent().setConstraint(this, layout);
  }

  public String getText() {
    return nameLabel.getText();
  }

  public Rectangle getTextBounds() {
    return nameLabel.getTextBounds();
  }

  public void setText(String text) {
    nameLabel.setText(text);
  }
}
```

Listing 158: Example of a figure using the defined interface

8.3.8.4.4 Controller

Introduction

Beside these two relatively independent model and view also a controller exists. In GEF it is called *EditPart* and acts as the mediator between these two responsible for managing the model, creating the view and to interact with GEF. This covers the registration of edit policies to handle user interactions, providing the required data for the complete visualization of the diagram as well as synchronizing the model with the view.

Implementation

For implementation of your edit parts GEF provides the interface `EditPart` with several abstract implementations of it. We recommend you to use the `AbstractGraphicalEditPart` respectively `AbstractConnectionEditPart` for the connections. They provide extensive default implementations, which can be adjusted or extended if required.

Like in other components of the graphical editor, too, we suggest you to locate common parts in an abstract super class. You can use synergies between the different edit parts. Otherwise, they had to be implemented in each edit part.

Node

As mentioned above, we recommend using the class `AbstractGraphicalEditPart` for the implementation of the edit parts, which represents nodes, labels or something like that. Through the default implementations, you only have to override the methods you really need.

Exceptions are the methods `createEditPolicies` and `createFigure`, because they are defined as abstract. In the first one you must register the edit policies that should apply on this edit part respectively the corresponding figure, which will be created in the second method. **Listing 159** shows a simple implementation of these two methods.

```
@Override
protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE,
      new NodeEditPolicy());
}

@Override
protected IFigure createFigure() {
  return new NodeFigure();
}
```

Listing 159: Simple example of the methods `createEditPolicies` and `createFigure`

Typical methods you will also override are `getModelChildren` and `refreshVisuals`. The first one returns the model elements, which should be shown as children inside the figure of this edit part. The `refreshVisuals` method allows you to update the corresponding figure, like shown in **Listing 160** below. We recommend you to use the interface for casting the figure, if you implemented them with interfaces. Otherwise this is obviously not possible.

```
@Override
protected void refreshVisuals() {
  INodeFigure figure = (INodeFigure) getFigure();
  GPersonAttribute model = (GPersonAttribute) getModel();

  figure.setBackgroundColor(model.getBackgroundColor());
```

```
  figure.setForegroundColor(model.getForegroundColor());
  figure.setLayout(model.getLayout());
  figure.setText(model.getName());
  figure.repaint();
}
```

Listing 160: Simple example of the method `refreshVisuals`

Further methods you may override are `getModelSourceConnections`, `getModelTargetConnections`, `getSourceConnectionAnchor` and `getTargetConnectionAnchor`. The first two return the model elements representing the connections starting respectively ending in this edit part.

The remaining methods define the visualization of the anchors. For this purpose GEF already provides three classes. One is the `ChopboxAnchor` for rectangular shapes, `LabelAnchor` additionally with a text and `EllipseAnchor` for elliptical shapes. Which one you should use depends on the shape of your figure. **Listing 161** shows an example implementation of these methods.

```
public ConnectionAnchor getSourceConnectionAnchor(
      ConnectionEditPart connection) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getSourceConnectionAnchor(
      Request request) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
      ConnectionEditPart connection) {
  return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
      Request request) {
  return new ChopboxAnchor(getFigure());
}
```

Listing 161: Simple example of the methods `getSourceConnectionAnchor` and `getTargetConnectionAnchor` returning an anchor

Connection

A little different and also slighter in its implementations are the connections. For its implementation we recommend you to use the `AbstractConnectionEditPart`, because it provides simplifications. The inheritance forces you to implement the method `createEditPolicies`.

Different is the `createFigure` method. The abstract class implements this method and returns a `PolylineConnection`. We recommend you only to override this method if you want to create another line or to configure the line visualization, for example with a different line end. **Listing 162** shows such a configuration, which adds a black arrow at the target end.

```
@Override
protected IFigure createFigure() {
  PolylineConnection figure =
      (PolylineConnection) super.createFigure();
  figure.setForegroundColor(ColorConstants.black);
```

```
  figure.setTargetDecoration(new PolygonDecoration());
  return figure;
}
```

Listing 162: Configuration of a `PolylineConnection` in the createFigure method

Notification

After these implementations the edit parts works fine. Unfortunately, it is not able to react on changes in the model classes. For this purpose, the counterpart implementation of the notification mechanism in the model classes is required.

Based on our recommendation in chapter 8.3.8.4.2 to use the *JavaBeans property change*, it is required to implement the interface `PropertyChangeListener` in the edit part. It forces the implementation of the method `propertyChange`. Depending on the changed property you can invoke `refreshVisuals`, `refreshChildren` or another refresh method.

Further, you must register and deregister the edit part on the corresponding model class. For this purpose GEF provides the two methods `activate` and `deactivate`. They are proposed for such an implementation, as shown in **Listing 163**.

```
@Override
public void activate() {
  super.activate();

  Node model = (Node) getModel();
  model.addPropertyChangeListener(this);
}

@Override
public void deactivate() {
  super.deactivate();

  Node model = (Node) getModel();
  model.removePropertyChangeListener(this);
}
```

Listing 163: Example implementation of the `activate` and `deactivate` method

8.3.8.4.5 EditPartFactory

Introduction

In the graphical editor only the model objects are set as its content. The same applies for the edit parts, which return in several methods only model objects for the creation of new elements.

For this reason, a component is required which creates for a given model object the corresponding edit part. GEF provides for this purpose the interface `EditPartFactory`. It's main and only purpose is to instantiate the edit parts at a central location. Otherwise, this must be done in all classes which return the model objects. But such a decentral creation is not suitable.

Implementation

For the implementation of the edit part factory you must use the interface `EditPartFactory`, which forces you to override the method `createEditPart`. It's your choice how you implement the instantiation of the edit parts. Reflexion is

possible as well as a simple comparison with `instanceof` operator and the instantiation with `new`.

Integration in the Graphical Editor

But this is only half the battle. The edit part factory must be registered in the graphical editor. Otherwise it is not available. **Listing 164** shows an excerpt out of the graphical editor.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  // EditPartFactory
  viewer.setEditPartFactory(editPartFactory);

  ...
}
```

Listing 164: Excerpt of the graphical editor to set the `EditPartFactory`

8.3.8.4.6 Hints

Figure interface

To allow a decoupling of the figure from all the classes it is used in, we recommend defining a corresponding interface for it. The big advantage of such a solution is the decoupling of the figure definition and its concrete implementation. The classes using the figure can work simply with the interface without knowing how it is implemented exactly. This is very useful for a generic implementation of the further components of the graphical editor.

Generic figures

We recommend you to implement your figures as generic as possible, to increase the reusability of them in your graphical editor. Typically, you can use them more than once. Otherwise, you implement duplicate code in a large scale, which is unnecessary.

But for some purposes this is not possible or too complex to implement. In such cases it's fully legitimate to develop an own figure for each edit part.

Generic EditPartFactory

You should also consider implementing your `EditPartFactory` in a very generic way. It allows you to extend or exchange the classes he can create dynamically without changing the source code. This is especially an interesting possibility if you intend to use Spring for the wiring of your graphical editor.

8.3.8.5 Commands

Intention

The command objects encapsulate the accepted and converted user interactions on the figures respectively the diagram. The reason for this design decision of the GEF developer is its integration in Eclipse's undo/redo functionality.

Implementation

The implementations of the command objects correspond widely with the command concept in Eclipse using the interface `IUndoableOperation`. Instead using this interface, you must inherit from the abstract class `Command`.

Now you can override only those methods you really need. This covers typically the methods `execute` and `undo` as well as `canExecute` and `canUndo`. Further methods can also be overridden, if necessary.

So you can see the way the command objects must be implemented is given by GEF. So the leeway to vary them is very limited.

8.3.8.5.1 Hints

Separation of check and execution

The abstract class `Command` allows the implementation of checks for the execution respectively undo in a separate method. We recommend you to do the checks exactly in this way. Beside a clear separation and a better maintainability, it allows also GEF to visualize if the command can be executed or not.

8.3.8.6 EditPolicies

Intention

The edit policy is an important component of the graphical editor. Its responsibility is the handling of the user interactions. This covers on the one hand the conversion of the requests and interactions in command objects, where they can be applied on the model objects, and on the other hand to provide the feedback, which is used for the visualization.

Implementation

The GEF developers defined for the implementation of the edit policies the interface `EditPolicy` and provided abstract classes for the different operation purposes.

Generally, we recommend you to use the abstract classes, because they are already prepared for the handling of different request types. So they must be extended only with the solution-specific parts, through overriding methods. Otherwise, you have to implement the whole interface definition.

Although these abstract classes force the implementation of different methods depending on their purpose, the concept behind them is very similar. Typically, you have to override methods to create command objects or in rare cases to set or get some values. To get the required data for the command objects, the edit policy classes provide the method `getHost` and `getHostFigure`. The former returns the edit part on which the policy is installed and the latter the corresponding figure.

From the edit part you can get the corresponding model object as well as its parent and root edit part and extensive information about its status, like the selection or focus.

Frequently used edit policies

Now we show and explain you some frequently used edit policies. Like mentioned above, their implementation is very similar.

- The first one is the `ComponentEditPolicy`. We recommend you to use it for handling mainly the delete request on a diagram element. For this purpose, you have to override the method `createDeleteCom-`

mand. You can get the element to delete through the `getHost` method or from the passed `GroupRequest`, which provides all selected edit parts. But we disadvise you from using the edit parts from the request. The reason for this recommendation is the fact, that all selected elements have an own edit policy in a normal case. Additionally, it would be necessary to implement either a command object handling the deletion of more than one element or a compound command containing an own command object for each element to delete.

- A further frequently used edit policy is the `ConnectionEditPolicy`. It's very similar to the `ComponentEditPolicy` but more limited on deleting connections. For this purpose, it provides the abstract method `getDeleteCommand`. Apart from that, the rest of the implementation is exactly equals.
  As an alternative it's also possible to use for the deletion of the connections the `ComponentEditPolicy`.

- Different is the `GraphicalNodeEditPolicy`. It's responsible for creating and reconnecting connections in the graphical editor. For this purpose, it provides four abstract methods, which must be implemented. The method `getConnectionCreateCommand` is invoked when a user begins to draw a connection. It must create a command object, set the source edit part and pass it with the method `setStartCommand` to the command object.
  The counter part is the method `getConnectionCompleteCommand`, which gets the command object from the passed request and set it the target edit part.
  A little different are the methods `getReconnectSourceCommand` and `getReconnectTargetCommand`. They must create and return also a new command object. Through the passed request you can get the edit part of the connection as well as the new source respectively target object through the `getHost` method.
  We recommend you to implement the command for creating respectively reconnecting the connections in a generic manner. So you can use it for both purposes and reduce the amount of code.

- To move the elements and change its size, we recommend you to use the `XYLayoutEditPolicy`, if you use the `XYLayout` in the parent element. Additionally, it's also responsible for creating new elements based on the click location.
  For this purpose, the two methods `createChangeConstraintCommand` and `getCreateCommand` must be implemented. They both return a command object. In the former one a constraint is passed, which is a `Rectangle` containing the new size and location of the element. In the latter one you can get the constraint (`Rectangle`) through the method `getConstraintFor` where you pass the request.

- A further frequently used edit policy is the `DirectEditPolicy`. Its responsibility is to show the direct edit feedback and to create the command to perform the direct edit. For this purpose the two abstract methods `getDirectEditCommand` and `showCurrentEditValue` must be implemented. In the former one you have to create a command object applying the direct edit. You get the edited value through the statement `request.getCellEditor().getValue()`.
  In the latter method you must provide the feedback of the edit. You get the edited value through the same statement and can pass it to the corresponding figure.

Example implementations for a better understanding of the described edit policies are shown in chapter 6.3.8.6 and 6.3.8.7.

Certainly, further edit policies for other purposes are provided by GEF. From the concept, they are very similar, so we renounce to describe them here in detail.

Registration

To allow the takeover of this role, the edit policies must be registered on the edit parts where they should apply to. We recommend you to do the registration in the method `createEditPolicies`, because this is the proposed location. You have to use the `installEditPolicy` method where pass an instance of the edit policy as well as an identifier used to key it. **Listing 165** shows a code snippet of such a registration.

```
@Override
protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE,
      new NodeEditPolicy());
  installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
      new ConnectionNodeEditPolicy());
  installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE,
      new NodeDirectEditPolicy());
}
```

Listing 165: Excerpt of an edit part displaying the edit policy registration

A detailed description of the possible identifiers is provided by the interface `EditPolicy`, so we refer to its Javadoc.

8.3.8.7 Direct Edit

Intention

The direct edit feature allows the user of the graphical editor to edit the elements directly without any wizards or dialogs. So it allows a very convenient use of the editor.

Structure

Its implementation is a bit more complex compared with other components and features of the graphical editor. The reason lies in its structure, shown in **Figure 43**, which covers several classes.

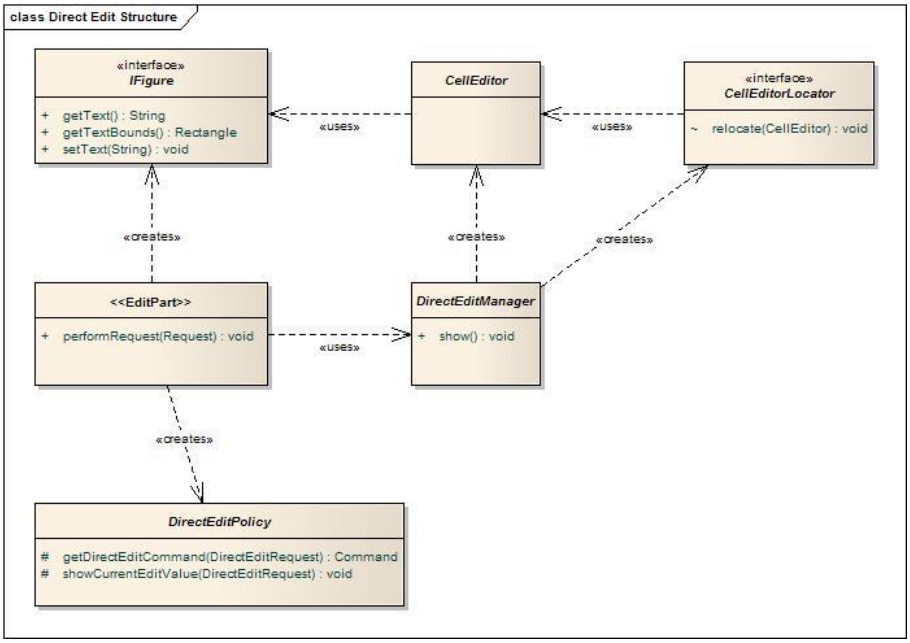

Figure 43: Schematic diagram of the classes involved in the direct edit

Beside the `DirectEditPolicy`, already described in chapter 8.3.8.6, also a `DirectEditManager` and a `CellEditorLocator` is required. Their key responsibilities are the management of the direct edit operation, the creation and the initialization of the `CellEditor`. The `CellEditor` implementations itself are provided by GEF.

Figure

Additionally, the figure must be extended to allow access to the value which should be edited as well as the location where the direct edit should be displayed. Otherwise, the visualization handling would be very complex and fragile.

For this purpose and based on the recommendation about the definition of an interface for the figures in chapter 8.3.8.4.3, we recommend you to define an interface providing the required methods for the direct edit. The interfaces of the concrete figures must extend instead of `IFigure` the new defined interface.

Listing 166 shows such an interface with the required methods for the direct edit.

```
public interface ITextFigure extends IFigure {

  public String getText();

  public void setText(String text);

  public Rectangle getTextBounds();
}
```

Listing 166: Definition of an interface with the methods required for the direct edit

Direct edit manager

Based on this interface, you can now develop a generic `DirectEditManager` implementation. If you don't have an interface or an abstract class for your figures, you have to implement an own `DirectEditorManager` for each of them

or one with several if-statements. But that's simply code duplication and makes no sense. Therefore, we disadvise to implement such a solution.

For its implementation, you must inherit from `DirectEditManager`, which forces you to implement the method `initCellEditor` and a constructor passing the edit part where it applies on, which kind of `CellEditor` you intend to use and an instance of the `CellEditorLocator`. We recommend passing the `CellEditor` type as a fix statement, because the remaining part of the direct edit manager is based on it. Therefore, an exchange leads inevitably to changes in class.

The following **Listing 167** shows an example of such a constructor implementation using the `TextCellEditor`.

```
public class TextDirectEditManager extends DirectEditManager {

  private class TextVerifyListener implements VerifyListener {

    public void verifyText(VerifyEvent event) {
      Text text = (Text) getCellEditor().getControl();
      String oldText = text.getText();
      String leftText = oldText.substring(0, event.start);
      String rightText = oldText.substring(event.end,
          oldText.length());

      GC gc = new GC(text);
      Point size = gc.textExtent(leftText + event.text +
                                      rightText);
      gc.dispose();

      if (size.x != 0) {
        size = text.computeSize(size.x, SWT.DEFAULT);
      }
      getCellEditor().getControl().setSize(size.x, size.y);
    }
  }

  protected ITextFigure textFigure;

  protected Font scaledFont;
  protected VerifyListener verifyListener;

  public TextDirectEditManager(GraphicalEditPart source,
      ITextFigure figure) {

    super(source, TextCellEditor.class,
        new TextCellEditorLocator(figure));
    this.textFigure = figure;
  }

  @Override
  protected void bringDown() {
    Font disposeFont = this.scaledFont;
    this.scaledFont = null;

    super.bringDown();

    if (disposeFont != null) {
      disposeFont.dispose();
    }
  }

  @Override
```

```
  protected void initCellEditor() {
    Text textControl = (Text) getCellEditor().getControl();

    verifyListener = new TextVerifyListener();
    textControl.addVerifyListener(verifyListener);

    getCellEditor().setValue(textFigure.getText());

    IFigure figure =
        ((GraphicalEditPart) getEditPart()).getFigure();
    scaledFont = figure.getFont();
    FontData data = scaledFont.getFontData()[0];
    Dimension fontSize = new Dimension(0, data.getHeight());
    data.setHeight(fontSize.height);
    scaledFont = new Font(null, data);

    textControl.setFont(scaledFont);
    textControl.selectAll();
  }

  @Override
  protected void unhookListeners() {
    super.unhookListeners();

    Text text = (Text) getCellEditor().getControl();
    text.removeVerifyListener(verifyListener);
    verifyListener = null;
  };
}
```

Listing 167: Example of a direct edit manager using a `TextCellEditor`

In the method `initCellEditor` it's up to you, what you want to configure. We propose to set the value out of the figure in the direct edit field at least. Otherwise, the user always has to rewrite the value if he only wants to extend it.

Further configurations which are possible are setting the same font as in the figure or the registration of a `TextVerifyListener`. Its purpose is to calculate the size of the direct edit field continuously to provide an optimal visibility. Please note that you should deregister the listener in the method `unhookListeners`, which can be overridden. Otherwise, the cleanup won't be applied properly. We recommend you to do them for an increased usability of your graphical editor, but it's up to you if you decide to do it or not.

Cell editor locator

Like you already noticed in **Listing 167**, it is also necessary to pass to the constructor of the `DirectEditManager` an instance of a cell editor locator. For this purpose, you have to implement it using the interface `CellEditorLocator`. The only method it forces to override is `relocate`, whose purpose is to relocate and resize the direct edit field of the passed `CellEditor`. An example for `CellEditor` using a `Text` control is shown in **Listing 168**.

```
public class TextCellEditorLocator implements CellEditorLocator {

  private ITextFigure textFigure;

  public TextCellEditorLocator(ITextFigure figure) {
    this.textFigure = figure;
  }

  public void relocate(CellEditor celleditor) {
```

```
    Text textControl = (Text) celleditor.getControl();
    Point preferredSize =
        textControl.computeSize(SWT.DEFAULT, SWT.DEFAULT);
    Rectangle textBounds = textFigure.getTextBounds();

    textControl.setBounds(textBounds.x - 1, textBounds.y - 1,
        preferredSize.x + 1, preferredSize.y + 1);
  }
}
```

Listing 168: Example of a text-based cell editor locator implementation

Edit part integration

After the implementation of all required parts for the direct edit, also the integration in the graphical editor is necessary. It consists of two different configurations. The first one is the registration of the direct edit policy in the method `createEditPolicies`, like shown in **Listing 169**.

```
@Override
protected void createEditPolicies() {
  ...
  installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE,
      new NodeDirectEditPolicy());
}
```

Listing 169: Code snippet out of the edit part integrating the `DirectEditPolicy`

The second one consists of the overriding of the `performRequest` method. Here, the `DirectEditManager` must be instantiated and shown. We recommend creating a new instance only in cases where no one exists yet. This increases the performance compared with a continuous instantiation.

We also recommend you to react not only to the direct edit request (single click), but also for the open request (double click). The reason for this recommendation is the fact a user typically makes a double click. So it increases the usability.

Listing 170 shows an excerpt out of the edit part overriding the `performRequest` method for the integration of the `DirectEditManager`.

```
@Override
public void performRequest(Request request) {

  if (request.getType() == RequestConstants.REQ_DIRECT_EDIT
   || request.getType() == RequestConstants.REQ_OPEN) {

    if (manager == null) {
      manager = new TextDirectEditManager(this,
          (INodeFigure) getFigure());
    }

    manager.show();
  }
}
```

Listing 170: Code snippet out of the edit part integrating the `DirectEditManager`

8.3.8.7.1 Hints

Common implementation

We recommend you to define a common interface for the figures you want to provide the direct edit feature. This allows you to implement several classes for this feature very generic, like the `DirectEditManager` or the `CellEditorLocator` shown in the listings above, which uses the `ITextFigure` interface.

Otherwise, you must implement on the one hand several if-statements in these classes or even own classes for each figure. Such a solution is very fragile and leads to the *Shotgun Surgery* smell *[Smells]*. That's why we disadvise you to implement

TextVerifyListener

Very useful for the continuous calculation of the required field size for the direct edit is the `TextVerifyListener`. During the input of text, it is invoked regularly and is therefore able to calculate the field. Although it's only a nice to have feature, we recommend you to implement it. Otherwise, the user cannot see the whole input of the field, because it changes its size only reluctantly.

Other cell editors

Beside the `TextCellEditor` GEF provides further ones with different ways to change the value. This covers the `ComboBoxCellEditor`, `CheckboxCellEditor` as well as a `DialogCellEditor`. They help to limit the values a user can select.

Generally, their implementation is very similar but can vary in selected locations. Please consider these circumstances.

8.3.8.8 Palette

Intention

The palette is a panel on the right or left side of the graphical editor. It provides the tools for drawing new elements and connections. Therefore, it is a very important feature for the users of the graphical editor. Alternative solutions to draw new elements are possible but have a worse usability compared with the palette or are extension of the palette itself.

For this reason, GEF provides extensive support for the configuration and implementation of the palette as well as its integration in the graphical editor. So its implementation is very compact and convenient.

Structure

We recommend you to implement the palette in an own class outside of the graphical editor itself. Together with an interface or an abstract super class, such a solution has the advantage of a more focused editor implementation. Additionally, it allows an easier exchange of the palette through substitutions. **Listing 171** shows an example for such an interface.

```
public interface PaletteFactory {

  public PaletteRoot createPaletteRoot();
}
```

Listing 171: Definition of an interface for the class creating the palette

Naturally it's also possible to implement the palette in the graphical editor by yourself. It works also fine, but has the disadvantage of a strong coupling be-

tween the palette and the graphical editor. That's the reason why we disadvise such a solution.

Palette

The implementation of the palette itself is very compact and straightforward. In a first step, a `PaletteRoot` must be created, which serves as the root for the whole palette model.

Afterwards, it's up to you to decide which tools you intend to add to the palette, if you want to group them in any way and what style the different groups should have. For this purpose, GEF provides a Composite *[GoF1995]* like structure with container and entry classes.

For the grouping of the tool entries this covers the `PaletteDrawer`, a collapsible container, the `PaletteGroup`, a default container, the `PaletteStack` and the `PaletteToolbar`, visualize the entries in a toolbar-like style. We recommend you to use mainly the `PaletteDrawer` and the `PaletteToolbar`, because they are the most flexible and user-friendly containers.

For the tool entries, GEF provides mainly the `CreationToolEntry` and its two derivations `CombinedTemplateCreationEntry` and `ConnectionCreationToolEntry`. The first one is for a typical tool entry, whereas the second class is required if you decided to support the drag and drop (see chapter 8.3.8.9). The latter is a tool entry specialized for the creation of connections. Which one you decide to use is up to you. We can recommend each of these tools, because their implementation is very convenient.

Listing 172 shows an example of such a palette implementation. Based on the recommendation above it's located in an own class.

```
public class PersonLocatorPaletteFactory implements PaletteFactory {

  public PaletteRoot createPaletteRoot() {
    PaletteRoot palette = new PaletteRoot();

    palette.add(createToolsGroup(palette));
    palette.add(createNodesGroup());

    return palette;
  }

  private PaletteEntry createNodesGroup() {
    PaletteDrawer toolbar = new PaletteDrawer("Nodes");

    toolbar.add(new CombinedTemplateCreationEntry("Statement",
        "Create a new statement", GStatement.class,
        new NodeCreationFactory(GStatement.class), null, null));
    toolbar.add(new CombinedTemplateCreationEntry("Person",
        "Create a new person", GPersonAttribute.class,
        new NodeCreationFactory(GPersonAttribute.class), null,
        null));
    toolbar.add(new CombinedTemplateCreationEntry("Location",
        "Create a new location", GLocationAttribute.class,
        new NodeCreationFactory(GLocationAttribute.class), null,
        null));
    toolbar.add(new CombinedTemplateCreationEntry(
        "Attribute value", "Create a new attribute value",
        GAttributeValue.class, new NodeCreationFactory(
        GAttributeValue.class), null, null));

    return toolbar;
```

```
  }

  private PaletteEntry createToolsGroup(PaletteRoot palette) {
    PaletteToolbar toolbar = new PaletteToolbar("Tools");

    // Selection tool
    toolbar.add(new PanningSelectionToolEntry());

    // Marquee tool
    toolbar.add(new MarqueeToolEntry());

    // Connection tool
    toolbar.add(new ConnectionCreationToolEntry("Connection",
        "Create a new connection", new EmptyCreationFactory(),
        PersonLocatorGraphPlugin
            .getImageDescriptor("icons/connection_s16.gif"),
        PersonLocatorGraphPlugin
            .getImageDescriptor("icons/connection_s24.gif")));

    return toolbar;
  }
}
```

Listing 172: Example of a palette implementation using different container and tool classes

Creation factory

As you have already noticed in the example shown above, in the tool entries there must be passed a factory instance. Its only purpose is to instantiate the new model object for the element which is drawn by the user. For its implementation, GEF provides the `CreationFactory` interface which forces you to override the two methods `getNewObject` and `getObjectType`. The first method must return a new instance of the object and the second one returns their type.

How you implement your `CreationFactory` it's up to you. We recommend you to develop a generic one, for example with reflexion, to reduce the code and increase the reusability.

Integration in the graphical editor

After the implementation of the palette and the factory, the last step is its integration in the graphical editor.

For this purpose GEF provides already `GraphicalEditor` implementations which support the configuration of the palette. This converts on the one hand the `GraphicalEditorWithPalette` with a fix palette and on the other hand the `GraphicalEditorWithFlyoutPalette` with a moveable palette. In both classes the method `getPaletteRoot` must be overridden, which returns the `PaletteRoot` for the creation of the palette. We recommend you to use one of these two classes for the palette implementation, because they are very convenient and allow a compact configuration.

The second step for the integration of the palette covers the extension of the edit parts where the new elements should be added to. If you don't have a `LayoutEditPolicy` yet, you have to register it and override its getCreateCommand method. Consequently, there is also a command object required.

8.3.8.8.1 Hints

Wiring with Spring

To wire your graphical editor properly with Spring you have to decide whether you want to implement the palette in the class itself or not. Generally, we recommend locating it in an own class with a common interface, as already shown in **Listing 171**. This allows you to exchange an implementation easily with a substitute, without adjusting the source code.

In such a case, you have to add an attribute to the graphical editor with the corresponding setting method. The injection itself is very simple.

8.3.8.9 Drag & Drop

Intention

An extension of the palette is the drag and drop feature. It allows the user to drag a tool entry from the palette to the diagram and to drop it there as a new element. Therefore, it allows an alternative way to draw new diagram elements.

Palette extension

Like mentioned in chapter 8.3.8.8, GEF provides for the tool entries in the palette the class `CombinedTemplateCreationEntry`. Beside the `CreationFactory` also a template object can be passed which is used for the creation of the new element through the drag and drop mechanism.

What kind of template object you want to pass is your choice. It's possible to pass a `CreationFactory`, an instance of class or its class object, whatever you want. But consider that you have to instantiate the new element with the passed template object in a later step.

We recommend you to pass a `CreationFactory`. The reason for this recommendation is the possibility to benefit from default implementations of the drop target listener, which uses the factory for the creation of the new element. Otherwise, you are forced to create an own implementation of the drop target listener or even to distinguish programmatically between the template objects for its creation.

Drop target listener

But this is only one side for the development of the drag and drop feature. Now it's required to implement listeners for the graphical editor which allow the creation of the elements. For this purpose, GEF provides together with Eclipse RCP the two interfaces `TransferDragSoruceListener` and the `TransferDropTargetListener`.

Fortunately, you have to implement only the drop target listener. For the drag source listener, a generic default implementation `TemplateTransferDragSourceListener` is already provided and utilized.

Also for the drop target listener GEF provides the class `TemplateTransferDropTargetListener` as a default implementation. If you've passed the `CreationFactory` as the template object in the palette, you are able to use this class without adaptations. Otherwise, you have to implement an own drop target listener. Fortunately, its development is very convenient, because through the inheritance from the provided class only the method `getFactory` must be overridden, s shown in **Listing 173** below.

So we recommend you to pass in the palette `CreationFactory` instances as the template objects to allow an efficient use of the default implementations.

```
public class NodeTemplateTransferDropTargetListener extends Tem-
```

```
plateTransferDropTargetListener {

  public NodeTemplateTransferDropTargetListener(EditPartViewer
viewer) {
    super(viewer);
  }

  @Override
  protected CreationFactory getFactory(Object template) {
    return new NodeCreationFactory(
        (Class<? extends Node>) template);
  }
}
```

Listing 173: Example implementation of a drop target listener

Graphical editor extension

The implementation and configuration of these listeners is not enough. Additionally, the drop target listener must be registered on the `GraphicalViewer` and the drag source listener on the `PaletteViewer`, like shown in **Listing 174** below. Otherwise, the functionality is not provides and cannot be used.

We recommend you to do this registration in the `initializeGraphicalViewer` method of the editor, because the required components are only here available, not before. We additionally recommend you to set register these listeners in the `dispose` method to allow a proper cleanup.

```
@Override
protected void initializeGraphicalViewer() {
  super.initializeGraphicalViewer();

  ...

  GraphicalViewer viewer = getGraphicalViewer();
  dropTargetListener =
     new TemplateTransferDropTargetListener(viewer);
  viewer.addDropTargetListener(dropTargetListener);

  PaletteViewer paletteViewer = getPaletteViewer();
  dragTargetListener =
     new TemplateTransferDragSourceListener(paletteViewer);
  paletteViewer.addDragSourceListener(dragTargetListener);
}
```

Listing 174: Registration of the drag source and drop target listener in the graphical editor

8.3.8.9.1 Hints

Drop target listener

We recommend you to pass in the palette a `CreationFactory` as the template object. This is because the default implementation in the `TemplateTransferDropTargetListener` can handle it. Otherwise, you must create a subclass of it an override the method `getFactory`.

To avoid such an additional class we recommend you to pass by default a `CreationFactory` as template object in the palette, or to consider at least this circumstance.

8.3.8.10 Outline

Intention

The outline view is an optional component of the graphical editor. It gives an overview about the elements displayed in the diagram but in a more structured way, mostly as a tree. Through the outline of the whole diagram it's possible for the use to find a specific element easier than in the diagram.

For its implementation, Eclipse RCP and GEF provides several base and abstract classes, which can be used.

Structure

But before we explain its development in detail, we want to give you some recommendations. In several tutorials and example implementations you will see the outline will be located as an inner class in the graphical editor. In our opinion, this is a bad design, why we recommend you to implement it in an independent class based on the comparison in chapter 6.3.8.10.1. Additional to more focused classes you get the chance to exchange the outline easily through a substitute, which is valuable especially when you use the dependency injection for example with Spring.

We also recommend you to implement the outline view as generic as possible. By such a solution it can be reused without adaptions for further graphical editors.

Outline

For the implementation of the outline class itself, GEF provides the `ContentOutlinePage`. It forces you to pass an `EditPartViewer` in the constructor. We recommend you to pass the `TreeViewer`, because it's the only alternative viewer to the `GraphicalViewer`.

As further methods you have to override and extend `createControl`, `dispose`, `getControl` and `init`, like shown in **Listing 175** below.

In the method `createControl`, a composite control must be created where the viewer can be displayed on. It is also required to set the `EditDomain` and an `EditPartFactory`, because the GEF developers decided to use the same concept with the edit parts also for the outline. Apart from the view and a more compact edit part inheriting from `AbstractTreeEditPart`, the implementation of these classes is the same. So we don't explain it here again.

It is also required to set the content which should be displayed in the outline as well as implementing the selection synchronization in this method.

```
public class DefaultOutlinePage extends ContentOutlinePage {

  protected SashForm sash;
  protected GraphicalEditor editor;
  protected EditPartFactory editPartFactory;

  public DefaultOutlinePage() {
    super(new TreeViewer());
  }

  @Override
  public void createControl(Composite parent) {
    sash = new SashForm(parent, SWT.VERTICAL);

    getViewer().createControl(sash);
    getViewer().setEditDomain(
        (EditDomain) getEditorResource(EditDomain.class));
    getViewer().setEditPartFactory(editPartFactory);
```

```
    updateContents();

    SelectionSynchronizer synchronizer =
        getEditorResource(SelectionSynchronizer.class);
    synchronizer.addViewer(getViewer());
}

  @Override
  public void dispose() {
    SelectionSynchronizer synchronizer =
        getEditorResource(SelectionSynchronizer.class);
    synchronizer.removeViewer(getViewer());

    super.dispose();
  }

  @Override
  public Control getControl() {
    return sash;
  }

  @SuppressWarnings("unchecked")
  protected <T> T getEditorResource(Class<?> resourceType) {
    T resource = null;

    if (editor.getAdapter(resourceType) != null) {
      resource = (T) editor.getAdapter(resourceType);
    }

    return resource;
  }

  @Override
  public void init(IPageSite pageSite) {
    super.init(pageSite);

    // Action bars, context menus, key handlers, etc.
    ...
  }

  ...

  public void updateContents() {
    getViewer().setContents(
        (Object) getEditorResource(Object.class));
  }
}
```

Listing 175: Example implementation of an outline view

The method `init` is after the creation of the outline view the appropriate location to extend its functionality with action bars, context menus, key handlers and further more.

Editor resources

Like you already noticed in the **Listing 175** above, the outline requires different classes and resources provided by the graphical editor. To get them, we recommend you to pass the graphical editor to the outline view when it is created. The graphical editor should additionally provide the resources in its `getAdapter` method.

The reason for such a recommendation is the fact that no additional interface for the graphical editor is required to provide the resources. Additionally, the Extension Interface Pattern *[Extension Interface06]* is designed for exactly such pur-

poses. So your implementation will be totally Eclipse RCP compliant.

Integration in the graphical editor

The real integration in the graphical editor is very simple. The only required implementation is to extend the `getAdapter` method. If the passed class type is equals `IContentOutlinePage` an instance of the outline view must be returned, like shown in **Listing 176** below

```
@Override
public Object getAdapter(Class type) {
  if (type == IContentOutlinePage.class) {
    return outlinePage;
  } else if (type == EditDomain.class) {
    return getEditDomain();
  } else if (type == SelectionSynchronizer.class) {
    return getSelectionSynchronizer();
  } else if (type == Object.class) {
    return contents;
  }

  return super.getAdapter(type);
}
```

Listing 176: Extension of the graphical editor for the integration of the outline view

Based on the recommendation above, you also have to return further resources like the `EditDomain`, the `SelectionSynchronizer` or the displayed content. Otherwise, you have to pass them to the outline when it is instantiated. But this is a less flexible solution, why we disadvise to implement it in this way.

8.3.8.10.1 Hints

Wiring with Spring

The wiring of the graphical editor opens the possibility to implement the corresponding outline fully against interfaces and abstract classes. This, because the concrete implementations can be injected.

Through the abstract super class `ContentOutlinePage` you don't even have to define a common interface. It is already possible to exchange the implementation with a substitute, without adjusting the source code.

You only have to add an attribute to the graphical editor with the corresponding setting method. The injection itself is very simple.

Notify the outline about changed content

If you have the situation the content of your graphical editor updated or changed from outside the component, you will get a problem with the outline, because it won't be updated. To solve this problem, we recommend you to define an interface or an abstract class extending your outline view. In this method, you can define a method for the notification, as shown in **Listing 177** below.

```
public abstract class ExtContentOutlinePage extends ContentOutli-
nePage {

  public ExtContentOutlinePage(EditPartViewer viewer) {
    super(viewer);
```

```
  }

  public abstract void updateContents();
}
```

Listing 177: Abstract class to extend the `ContentOutlinePage` with common methods

In your graphical editor you can now work against this interface respectively abstract class and no longer against `OutlineContentPage`.

8.3.8.11 Miniature View

Intention

The miniature view is typically an extension of the outline view, because it provides a different kind of the outline visualization. It gives the user the possibility to get a small overview about the whole diagram as well as the currently shown selection. This increases the usability and allows an alternative way to scroll in the diagram.

Naturally it's also possible to implement it as an own page outside the outline view, but of its outlining purpose it's the obvious location. For that reason, we recommend to integrate it in the outline view.

Outline extension

Unfortunately, GEF provides for its implementation only the base classes not a default component or something similar. One of it is for example the `ScrollableThumbnail`.

Therefore, it's required to override the methods `createControl` and `dispose` in the outline view. In the former one you have to create a new control where the miniature view is displayed. Afterwards, the root edit part from the `GraphicalViewer` must be got to create the `ScrollableThumbnail`. This thumbnail must be added afterwards to the control you've created before.

For a proper cleanup you should additionally register a `DisposeListener`, which must be deregistered in the `dispose` method. Otherwise, the thumbnail cannot be disposed properly.

The **Listing 178** shows an example of such an extended outline using the mentioned classes.

```
public class DefaultOutlinePageWithMiniatureView extends Defaul-
tOutlinePage {

  private ScrollableThumbnail thumbnail;

  private DisposeListener disposeListener = new DisposeListener()
  {
    public void widgetDisposed(DisposeEvent e) {
      if (thumbnail != null) {
        thumbnail.deactivate();
        thumbnail = null;
      }
    }
  };

  @Override
  public void createControl(Composite parent) {
```

```
    super.createControl(parent);

    // Miniature view
    Canvas canvas = new Canvas(sash, SWT.BORDER);
    LightweightSystem system = new LightweightSystem(canvas);

    GraphicalViewer viewer =
        getEditorResource(GraphicalViewer.class);
    ScalableRootEditPart rootEditPart =
        (ScalableRootEditPart) viewer.getRootEditPart();

    thumbnail = new ScrollableThumbnail(
        (Viewport) rootEditPart.getFigure());
    thumbnail.setSource(rootEditPart
        .getLayer(LayerConstants.PRINTABLE_LAYERS));

    system.setContents(thumbnail);

    viewer.getControl().addDisposeListener(disposeListener);
  }

  @Override
  public void dispose() {
    GraphicalViewer viewer =
        getEditorResource(GraphicalViewer.class);

    if (viewer.getControl() != null &&
       !viewer.getControl().isDisposed()) {
      viewer.getControl().removeDisposeListener(disposeListener);
    }

    super.dispose();
  }
}
```

Listing 178: Extension of the outline with a miniature view

Through the implementation in the outline view, no further integration in the graphical editor is required.

8.3.8.12 Action Bar

Intention

The action bar component of the graphical editor has two different purposes. The first one is to link the editor with global defined actions and secondly to configure editor specific action bars. This is required if the editor requires very specific actions in the tool and menu bar, which should be shown only when it's open.

For this reason, GEF provides the possibility to define an own action bar contributor for an editor. The abstract base class for its implementation is `ActionBarContributor`.

Accessing global action bar entries

To allow the usage of globally defined actions, you must override the method `declareGlobalActionKeys`. With the `addGlobalActionKey` the ids of the actions can be passed, which should be linked with the editor. The rest is handled by the abstract super class.

```
@Override
```

```
protected void declareGlobalActionKeys() {

  addGlobalActionKey(ActionFactory.UNDO.getId());
  addGlobalActionKey(ActionFactory.REDO.getId());
  addGlobalActionKey(ActionFactory.SAVE.getId());
}
```

Listing 179: Linking global actions to allow their use in the graphical editor

Editor specific action bars

Different are the editor specific action bars. For their composition two steps are required. In the first one, the actions must be created in the method `buildActions`, like shown in **Listing 180** below.

```
@Override
protected void buildActions() {
  addRetargetAction(new DeleteRetargetAction());

  addRetargetAction(new UndoRetargetAction());
  addRetargetAction(new RedoRetargetAction());

  addRetargetAction(new ZoomInRetargetAction());
  addRetargetAction(new ZoomOutRetargetAction());
}
```

Listing 180: Creation of actions for the editor specific action bars

After the creation of the actions, they must be contributed to the action bar, for example the toolbar. For this purpose, you must override the corresponding `contributeToXXX` method, where you can add the actions easily to the passed action bar manager. **Listing 181** shows such an implementation.

```
@Override
public void contributeToToolBar(IToolBarManager toolBarManager) {

  toolBarManager.add(getAction(ActionFactory.DELETE.getId()));

  toolBarManager.add(new Separator());

  toolBarManager.add(getAction(GEFActionConstants.ZOOM_IN));
  toolBarManager.add(getAction(GEFActionConstants.ZOOM_OUT));
  toolBarManager.add(new ZoomComboContributionItem(getPage()));
}
```

Listing 181: Contribution of actions to the editor specific toolbar

Integration in the graphical editor

The integration in the graphical editor can be done in the extension point `org.eclipse.ui.editors`. In the entry representing the graphical editor the contributor class can be configured for the id `contributorClass`. That's all; nothing more is must be done.

8.3.8.12.1 Hints

Missing activa-

Through the definition of actions in the global action bar it's possible to use

tion of global actions

them with links to graphical editor actions. But if they are not registered properly, they won't be active. To prevent this problem, you must register them in the application specific `ActionBarAdvisor`. Otherwise, this step must be done in the graphical editor itself.

But this is bad design to register the in the graphical editor but to define its action bars in a separate class. Therefore, we recommend you to do the registration of the global actions in application wide `ActionBarAdvisor`.

Using extension points

The definition of the graphical editor's action bars using extension points does not work, why we disadvise to try it. The programmatic way seems to be the only way for its configuration. This is also providing through the different example applications where the developers used only the programmatic way.

8.3.8.13 Context Menu

Intention

The context menu provides an alternative way for the user of the graphical editor. It's possible to add actions applying on concrete elements, like the delete, but also general ones for example to zoom in and out.

Implementation

For its implementation, we recommend, as for other components too, its location in an own class outside of the graphical editor. This has the advantage of an easy exchangeability of different implementations, especially in cases where dependency injection is used. Additionally, the configuration of the context menu for the graphical editor and its outline can be done in the same class. Because of that, it's possible to reduce duplicate code.

ContextMenu-Provider

This is generally supported by GEF through the abstract base class `ContextMenuProvider`. In its constructor, the `EditPartViewer` must be passed, where the context menu should be depending on.

We recommend you to pass also the `ActionRegistry`, where you can get the already defined actions. Otherwise, you must create the actions on your own before you can add it to the context menu.

For the real composition of the context menu, the `ContextMenuProvider` forces you to override the method `buildContextMenu`. Now you can get the actions from the `ActionRegistry` and add it simply to the passer `IMenuManager`, like shown in **Listing 182** below.

```
@Override
public void buildContextMenu(IMenuManager menu) {
  IAction action;

  GEFActionConstants.addStandardActionGroups(menu);

  action = actionRegistry.getAction(GEFActionConstants.ZOOM_IN);
  menu.appendToGroup(GEFActionConstants.GROUP_VIEW, action);

  action = actionRegistry.getAction(GEFActionConstants.ZOOM_OUT);
  menu.appendToGroup(GEFActionConstants.GROUP_VIEW, action);

  action = actionRegistry.getAction(ActionFactory.UNDO.getId());
  menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);
```

```
  action = actionRegistry.getAction(ActionFactory.REDO.getId());
  menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);

  action =
      actionRegistry.getAction(ActionFactory.DELETE.getId());
  menu.appendToGroup(GEFActionConstants.GROUP_EDIT, action);
}
```

Listing 182: Composition of the context menu in the `ContextMenuProvider` implementation

Integration in the graphical editor

Its integration in the graphical editor is very easy. The GEF developer proposes the method `configureGraphicalEditor` to such configurations. So you can pass an instance of your `ContextMenuProvider` implementation the `GraphicalEditor`'s `setContextMenu` method, like shown in **Listing 183** below.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  ...

  // Context Menu
  viewer.setContextMenu(new DefaultContextMenuProvider(
      viewer, getActionRegistry()));
}
```

Listing 183: Integration of the context menu in the graphical editor

Almost the same is the integration in the outline view. But you should do it in the `init` method instead. The remaining implementations are equals.

8.3.8.13.1 Hints

Using extension points

Like the definition of the graphical editor's action bars, also the context menu using extension points does not work, why we disadvise to try it. The programmatic way seems to be the only way for its configuration. This is also proven through the different example applications where the developers used only the programmatic way.

Wiring with Spring

To wire the graphical editor, it's possible to define also the context menu as an exchangeable part. Different than the outline or other parts, it's required to pass the `GraphicalViewer` and possibly further resources of the graphical editor in the constructor. For this reason, you must create a man in the middle which is responsible for the creation of the `ContextMenuProvider`. This is required, because Spring has no access to the resources of the graphical editor.

We recommend you to solve this problem with the definition an additional interface or abstract class. It should contain a method with the required parameters for creating the `ContextMenuProvider`, like shown in **Listing 184** below.

```
public interface ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
      EditPartViewer viewer, ActionRegistry registry);
}
```

Listing 184: Definition of an interface for the creation of the `ContextMenuProvider`

In the class implementing this interface you can now create a new instance of your `ContextMenuProvider` with the passed parameters, like shown in **Listing 185**.

```
public class DefaultContextMenuProviderFactory implements Con-
textMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
      EditPartViewer viewer, ActionRegistry registry) {
    return new DefaultContextMenuProvider(viewer, registry);
  }
}
```

Listing 185: Implementation of the interface for the `ContextMenuProvider` creation

In the Spring application context, you can now easily create an instance of this class and inject it in the graphical editor, where he can invoke the defined method.

8.3.8.14 KeyHandler

Intention

The key handlers allow the user to apply a selected choice of actions on the diagram and its elements with keystrokes. Compared with other interaction ways, like the toolbar or the context menu, it's by far the fastest way to apply actions. For this reason, it's important to provide the user keystrokes for the frequently used actions.

Implementation

GEF supports the implementation of key handlers for graphical editors with several classes. But before we start to explain how to implement and configure them we want to define the implementation structure.

We recommend you to implement the key handlers in an own class outside the graphical editor respectively the outline view. Together with the definition of an interface, it allows an easier exchange with a substitute through the lower coupling. This is especially interesting if you plan to use Spring for wiring the application in later step. **Listing 186** shows an example of such an interface.

```
public interface KeyHandlerFactory {

  public KeyHandler createKeyHandler(GraphicalEditor editor);

  public KeyHandler createOutlineKeyHandler(ActionRegistry regi-
stry);
}
```

Listing 186: Definition of an interface for the key handler implementation

Certainly, it's also possible to implement the key handlers directly in the graphical editor class respectively the one for the outline. But you should to consider in such a case you have higher coupling.

Key handler configuration

The composition and configuration of the key handlers itself are very convenient and straightforward. In a first step you must instantiate a `KeyHandler` object. If the key handler is intended for a graphical editor, we recommend you to use the `GraphicalViewerKeyHandler`. Compared with the conventional `KeyHandler`, he provides default keystrokes for the navigation in the graphical editor.

Afterwards, you can simple add keystrokes to the `KeyHandler` together with the corresponding action you can get from the `ActionRegistry`.

```
public class DefaultKeyHandlerFactory implements KeyHandlerFactory {

  public KeyHandler createKeyHandler(GraphicalEditor editor) {
    GraphicalViewer viewer = (GraphicalViewer) editor
        .getAdapter(GraphicalViewer.class);
    ActionRegistry registry = (ActionRegistry) editor
        .getAdapter(ActionRegistry.class);

    KeyHandler keyHandler =
        new GraphicalViewerKeyHandler(viewer);

    keyHandler.put(KeyStroke.getPressed(SWT.DEL, SWT.DEL, 0),
        registry.getAction(ActionFactory.DELETE.getId()));
    keyHandler.put(KeyStroke.getPressed('+', SWT.KEYPAD_ADD, 0),
        registry.getAction(GEFActionConstants.ZOOM_IN));
    keyHandler.put(KeyStroke.getPressed('-', SWT.KEYPAD_SUBTRACT,
        0), registry.getAction(GEFActionConstants.ZOOM_OUT));

    return keyHandler;
  }

  public KeyHandler createOutlineKeyHandler(ActionRegistry registry) {

    KeyHandler keyHandler = new KeyHandler();
    keyHandler.put(KeyStroke.getPressed(SWT.DEL, SWT.DEL, 0),
        registry.getAction(ActionFactory.DELETE.getId()));

    return keyHandler;
  }
}
```

Listing 187: Example of a `KeyHandler` implementation for a graphical editor and its outline

Integration in the graphical editor

Very similar to the context menu is also the integration of the `KeyHandler` in the graphical editor. The proposed location for it is the `configureGraphicalViewer` method. Here, it can be passed as parameter in the `GraphicalViewer`'s method `setKeyHandler`, like shown in **Listing 188** below.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();
```

```
  GraphicalViewer viewer = getGraphicalViewer();

  ...

    // KeyHandler definition
    viewer.setKeyHandler(
        keyHandlerFactory.createKeyHandler(this));
    }
```

Listing 188: Integration of the `KeyHandler` in the graphical editor

The same integration can be done in the outline view, but in the `init` method instead.

8.3.8.14.1 Hints

Wiring with Spring

Through the wiring of the graphical editor, it's possible to outsource the key handler creation in an own class like proposed in the paragraphs above. For this purpose, you must define a common interface or an abstract class like already shown in **Listing 186**. This allows implementing the graphical editor fully against this interface without knowing which concrete implementation is used.

So you can exchange the implementations with substitutes without changing the source code, because all the classes have to correspond with the defined interface.

Additionally, you only have to add an attribute to the graphical editor with the corresponding setting method. The injection itself is very simple.

8.3.8.15 ZoomManager

Intention

An important feature the user of a graphical editor is the zoom functionality. It allows the user to choose his preferred focus for a convenient and rapid work.

Implementation

GEF supports the implementation and integration of a zoom manager in a graphical editor with several classes, like the `ZoomManager`, to mention only one. But before we start to explain how to implement and configure them, we want to define the implementation structure.

We recommend you to implement the composition of the zoom manager in an own class outside the graphical editor. Together with the definition of an interface, it allows an easier exchange with a substitute through the lower coupling. This is an interesting aspect especially if you plan to use Spring for wiring the application in a later step. **Listing 189** shows an example of such an interface.

```
public interface ZoomManagerFactory {

  public ScalableRootEditPart createZoomManager(GraphicalEditor
editor);
}
```

Listing 189: Definition of an interface for the zoom manager implementation

Certainly, it's also possible to implement the zoom manager directly in the

graphical editor class. But you should to consider in such a case that you have higher coupling and for its change or extension always the editor class must be adjusted.

Creation

The composition and configuration of the zoom handler itself are very convenient and straightforward. To use this functionality, it's required to create in a first step an instance of the `ScalableRootEditPart`. It will be the new root element instead and contain the diagram as a child. Otherwise, the zoom functionality must be implemented by the edit parts and its figures itself which is cumbersome and leads to the *Shotgun Surgery* smell *[Smells]*.

From the `ScalableRootEditPart` you can get now the `ZoomManager`, where you can set the zoom levels, some zoom contributions, like shown in **Listing 190** below.

```
public ScalableRootEditPart createZoomManager(GraphicalEditor
editor) {

  ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
  ZoomManager manager = rootEditPart.getZoomManager();

  manager.setZoomLevels(new double[] { 0.25, 0.5, 0.75, 1.0, 1.5,
      2.0, 2.5, 3.0, 4.0, 5.0, 10.0, 20.0 });

  ArrayList<String> zoomContributions = new ArrayList<String>();
  zoomContributions.add(ZoomManager.FIT_ALL);
  zoomContributions.add(ZoomManager.FIT_HEIGHT);
  zoomContributions.add(ZoomManager.FIT_WIDTH);
  manager.setZoomLevelContributions(zoomContributions);

  registerZoomManagerActions(editor, manager);
  registerZoomMouseHandler(editor);

  return rootEditPart;
}
```

Listing 190: Composition of a `ZoomManager` for the graphical editor

Configuration

Additionally, you can do some further configurations of the `ZoomManager`. This covers the registration of actions and mouse handlers. We recommend the use of them, because they increase the usability and allow thereby an easier handling.

The registration of the zoom-in and zoom-out actions can be done with the `ActionRegistry` available through the `GraphicalEditor`'s `getAdapter` method, like shown in **Listing 191** below.

Also the configuration of the mouse wheel handler is very simple. It must be registered on the `GraphicalViewer` also available through the `GraphicalEditor`'s `getAdapter` method.

```
private void registerZoomManagerActions(GraphicalEditor editor,
      ZoomManager manager) {
  ActionRegistry registry = (ActionRegistry) editor
      .getAdapter(ActionRegistry.class);

  registry.registerAction(new ZoomInAction(manager));
  registry.registerAction(new ZoomOutAction(manager));
```

```
}

private void registerZoomMouseHandler(GraphicalEditor editor) {
  GraphicalViewer viewer = (GraphicalViewer) editor
      .getAdapter(GraphicalViewer.class);

  viewer.setProperty(MouseWheelHandler.KeyGenerator
      .getKey(SWT.SHIFT), MouseWheelZoomHandler.SINGLETON);
}
```

Listing 191: Configuration of the `ZoomManager` with actions and a mouse wheel handler

Integration in the graphical editor

The integration of the zoom manager in the graphical editor is very similar to the context menu and the key handler. The proposed location for it is the `configureGraphicalViewer` method. It can be passed as parameter in the `GraphicalViewer`'s method `setRootEditPart`, like shown in **Listing 192** below. Otherwise, the edit part of the displayed diagram will be the root edit part.

```
@Override
protected void configureGraphicalViewer() {
  super.configureGraphicalViewer();

  GraphicalViewer viewer = getGraphicalViewer();

  ...

    // ZoomManager definition
    viewer.setRootEditPart(
        zoomManagerFactory.createZoomManager(this));
}
```

Listing 192: Integration of the `ZoomManager` in the graphical editor

8.3.8.15.1 Hints

Wiring with Spring

Through the wiring of the graphical editor, it's possible to outsource the zoom manager creation and configuration in an own class like proposed in the chapter above. For this purpose, you must define a common interface or an abstract class like already shown in **Listing 189**. This allows implementing the graphical editor fully against this interface without knowing which concrete implementation is used.

So you can exchange the implementations with substitutes without changing the source code, because all the classes have to correspond with the defined interface.

Additionally, you only have to add an attribute to the graphical editor with the corresponding setting method. The injection itself is very simple.

8.3.9 Text Editor

Intention

The text editor provided in the Eclipse text facility is a powerful tool providing a lot of functions and configuration possibilities. It makes it an easy to use and

efficient tool for text processing operations. It is a very important contender in an application if somehow text based information has to be showed, parsed or processed.

In this chapter we show you how you can create and configure your own text editor implementation. We give you advices and recommendations based on our research in books and the internet and on our experiences gained during the implementation of the text editor.

Diagram

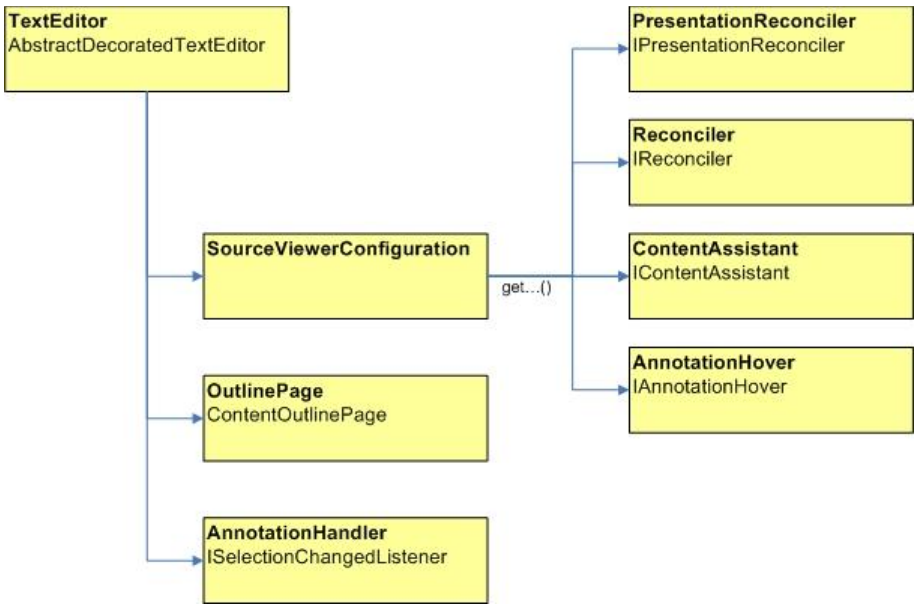

Figure 44: Structure of the text editor implementation

Structure

Referring to **Figure 44**, which shows the implementation structure of the text editor, it is apparent that some of the functionality is implemented inside of the `SourceViewerConfiguration` and some in the `TextEditor` itself.

The framework specifies in the most cases a default way of implementing the features, so that there is not much space for own, alternative implementations. However, there is still a large amount of variations available on how a text editor can be configured considering its behavior and the provided features. This configuration highly depends on the scope and the requirements of the application to be developed. Therefore, in the next chapters, we will describe our recommendation of the implementation of the text editor considering the scope of this bachelor thesis. Additionally, our recommendations are based on RCP best practices and are given by considering the Javadoc documentation.

8.3.9.1 Syntax highlighting

Overview

Syntax highlighting is a helpful tool when used in a structured document. It helps recognizing the structure of the document much faster and makes the text more readable.

Diagram

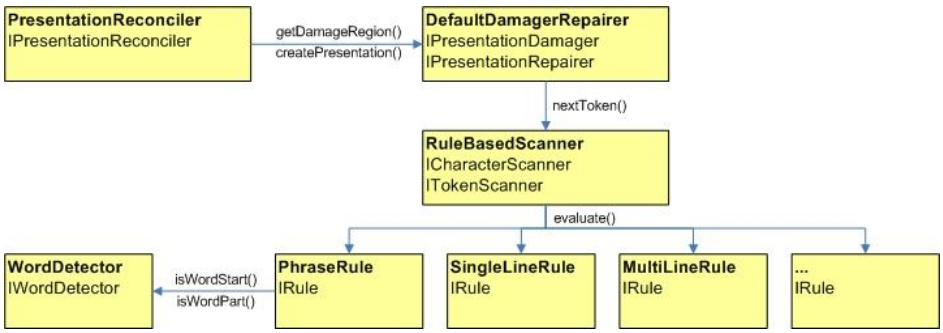

Figure 45: Diagram showing the structure of the text editor implementation

Point of extension

The implementation of the syntax highlighting is done through the extension of the `SourceViewerConfiguration` by overriding the method `getPresentationReconciler` and by returning an `IPresentationReconciler`.

Damager and Repairer

As pictured in **Figure 45** above, the presentation reconciler uses an `IPresentationDamager` and an `IPresentationRepairer` to determine the damaged region and create its presentation. Here, we recommend using a `DefaultDamagerRepairer`, if no special behavior is required for determination of the damaged region and the creation of the text presentation. The `DefaultDamagerRepairer` is a default implementation for syntax driven documents provided by the framework and covers both, the damager and the repairer.

The **Listing 193** below shows such a configuration.

```
public IPresentationReconciler getPresentationReconciler(
        ISourceViewer sourceViewer) {

  PresentationReconciler reconciler =
      new PresentationReconciler();
  DefaultDamagerRepairer dr =
      new DefaultDamagerRepairer(getScanner());

  reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
  reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

  return reconciler;
}
```

Listing 193: Example showing the setup of the damager and the repairer

Token scanner

In order to be able to scan the document and determine the tokens required to be able to create the presentation, the repairer requires an `ITokenScanner`. If the tokens in your document can be determined based on rules, we recommend using the `RuleBasedScanner` provided by the framework. It is a configurable scanner which uses an array of rules (`IRule`) to determine the tokens, as shown below.

```
private ITokenScanner getScanner() {
  RuleBasedScanner scanner = new RuleBasedScanner();

  IRule[] rules = new IRule[2];
  rules[0] = createPersonsWordRule();
```

```
  rules[1] = createLocationWordRule();

  scanner.setRules(rules);
  return scanner;
}
```

Listing 194: Example showing the implementation of the scanner and the rules

Rules

Referring to the rules, the framework provides default `IRule` implementations like `NumberRule`, `WordRule`, `SingleLineRule`, `MultiLineRule` and `WhitespaceRule`. By using these default rule implementations, a large set of token types in a document can be recognized.

For the recognition of numeric values as tokens, the use of `NumberRule` is recommended. It simply can be created by specifying the presentation of the token (`IToken`) using `TextAttribute` and attached on the token scanner to be active.

If single words, separated by whitespaces, have to be recognized, we recommend using the `WordRule`. The `WordRule` requires an implementation of `IWordDetector`, which specifies the word start and the word part characters. After the definition of the word detector, words can be added to the `WordRule` including the related specification of the token presentation (`TextAttribute`).

Having a text which can be separated based on start and end sequences, like e.g. javadoc comment in java source files, it is recommended to use `SingleLineRule` for tokens in a single line and `MultiLineRule` if the token goes over multiple lines. These types of rules can be defined by specifying the start and the end sequence of the delimiter as well as the description of the text presentation (`TextAttribute`).

The `WhitespaceRule` is capable of detecting whitespaces with the use of an `IWhitespaceDetector`, which specifies the characters to be considered as whitespaces. We recommend using the `WhitespaceRule` in cases, where a custom definition of what a whitespace is needs to be specified.

If the requirements of your text editor are to recognize whole phrases including whitespaces, like e.g. names of persons, inside of a text document, we recommend creating an own implementation of `IRule`. The best way to do this is taking a similar approach as the `WordRule` implementation, which uses an `IWordDetector` to detect words. In the case of phrase detection the `IWordDetector` should consider whitespaces as word parts in the `isWordPart` method. Afterwards, the `IWordDetector` can be used in the custom implementation of `IRule`.

The **Listing 195** below shows an example for the creation of rules.

```
private IRule createLocationWordRule() {
  IToken titleToken =
      new Token(new TextAttribute(
         fColors.getColor(new RGB(100, 100, 200)),
         null, SWT.BOLD)
      );

  PhraseRule phraseRule =
      new PhraseRule(new SimpleWordDetector(),
          titleToken, new ModelDataProvider(),
          ModelDataProvider.LOCATIONS);
```

```
  return phraseRule;
}
```

Listing 195: Example showing the creation of the phrase rules

General recommendation

We generally recommend using the default processes and implementations of the framework as long as the required functionality is provided. Additionally, we recommend creating own implementations only where custom behavior is required and where the framework provides extension points. This approach helps reducing the implementation effort and staying compatible with the philosophy of the framework and future framework changes.

Alternatives

As already mentioned above, the framework does not leave very much space for own, alternative solutions, because it defines the extension points and the implementation processes very strictly. However, there is still the possibility to specify the behavior and the configuration of the text editor.

Having custom requirements concerning the behavior for the determination of the damaged region and the creation of the text presentation, it can be reasonable to create your own `IPresentationDamager` and `IPresentationRepairer` implementations. We only recommend the own implementation of these components in circumstances, where the framework does not provide sufficient functionality in the default implementations.

Another alternative could be the implementation of an own `ITokenScanner`, if a custom behavior concerning the scan of a document and the determination of tokens is required.

If the requirements of your application, concerning the text scanning, can be fulfilled based on rules, the use of the `RuleBasedScanner` is recommended. With the definition of own implementations of `IRule` to specify the document partitioning, a flexible solution can be created without a huge implementation effort.

8.3.9.2 Model reconciliation

Overview

The RCP framework provides a reconciling facility, which allows updating the application model on changes in the text editor model. The text editor model (`IDocument`) changes on every user operation within the text in the document. Thus, an appropriate strategy for the model reconciliation has to be chosen depending on the requirements of the application to be developed.

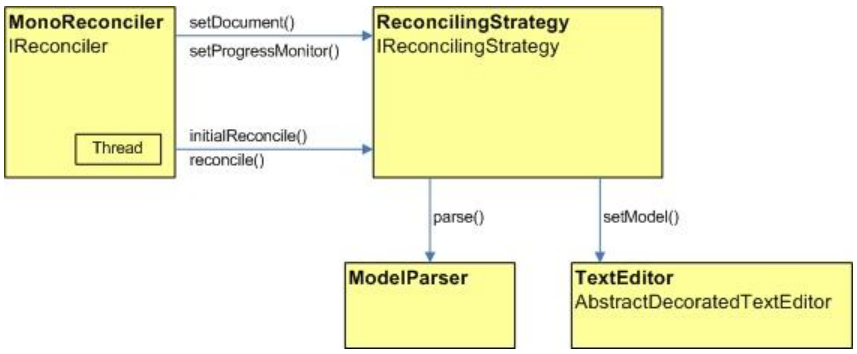

Figure 46: Diagram of the reconciling structure in the text editors configuration

Point of extension

The implementation of the reconciling is done through the extension of the `SourceViewerConfiguration` by overriding the method `getReconciler` and by returning an `IReconciler`.

Reconciler

The RCP framework provides `MonoReconciler` and `Reconciler` as default implementations of `IReconciler`. The `MonoReconciler` supports only one `IReconcilingStrategy` for the whole document, independent from the content type. The `Reconciler` supports multiple `IReconcilingStrategies` for different content types in the document. It is recommended to use the `MonoReconciler` if the requirements in your application do not specify document partitioning and different reconciling strategies for different content types (like e.g. comments, code, default content, etc.). Otherwise, the use of the `Reconciler` is more appropriate.

Progress monitor

Both reconcilers support the use of a progress monitor (`IProgressMonitor`), which can be set using the method `setProgressMonitor`. The progress monitor allows monitoring and controlling the progress of an activity.

Configuration

Referring to the `IReconcilingStrategy`, an own implementation has to be created and set on the reconciler. If you need to be able to use a progress monitor (`IProgressMonitor`) and perform initial actions before the reconciling starts (`initialReconcile`), the additional implementation of an `IReconcilingStrategyExtension` in your strategy implementation is required.

The reconciler notifies on every change the reconciling strategy. To update the application model, the use of an appropriate parser is recommended. Please note that the parser should be a completely RCP framework independent component, which is implemented based on the requirements of the application to be developed. The framework simply provides hook points to make it possible to react on document changes in the text editor.

After the parsing of the text in the document is done, it is recommended to pass the derived model to the text editor, if the text editor needs access on the application model, like for example to show the outline page or for other purposes.

To avoid starting the parsing process every time a key is pressed, it is recommended to set a delay for the reconciling activation. The `MonoReconciler` and the `Reconciler` both provide a method called `setDelay`, which allows the setting of an activation delay in milliseconds.

```
public IReconciler getReconciler(ISourceViewer sourceViewer) {
  PersonLocatorReconcilingStrategy strategy =
      new PersonLocatorReconcilingStrategy(fEditor);

  MonoReconciler reconciler =
      new MonoReconciler(strategy, false);
  reconciler.setProgressMonitor(new NullProgressMonitor());
  reconciler.setDelay(500);

  return reconciler;
}
```

Listing 196: Example showing the definition of a reconciling strategy

Alternatives

A possible alternative solution is to implement your own `IReconciler`. The default implementations in the framework covering `MonoReconciler` and `Reconciler`, are both based on a single background thread, which runs the reconciling activity. By implementing your own reconciler, custom and additional functionality could be provided. We only recommend such an individual implementation, if the provided default implementation in the RCP framework is not sufficient for your application requirements.

8.3.9.3 Outline page

Overview

An outline page is used to give the user an overview of the document structure, which is, especially in large documents, a very helpful tool. The RCP framework provides by default an outline view which can be used to display custom outline pages referring to the actual opened editor.

Point of extension

The RCP framework requests, every time an editor is opened, using the method `getAdapter` an `IContentOutlinePage` on the text editor and thus giving the possibility to return an own implementation of an outline page for your editor.

Concept

For the implementation of a common tree-based content page, the RCP framework provides the abstract class `ContentOutlinePage`, which implements an empty `TreeViewer` control for the model presentation. By extending this class to create an own implementation, the `TreeViewer` can be used to fill in the model data. Through the method `setInput` the input data for the `TreeViewer` can be set. The `TreeViewer` then uses an `IContentProvider` and an `IBaseLabelProvider` to get and illustrate the input data. The `IContentProvider` is responsible to return the model data in a tree-based form to be shown in the `TreeViewer`, whereas the `IBaseLabelProvider` returns the text and the icons of the elements in the tree.

Implementation

For the implementation of an outline page we recommend creating a subclass of `ContentOutlinePage` and using the default `TreeViewer` provided through the `getTreeViewer` method in the super class. The setup of the `TreeViewer` is done in the overridden `createControl` method of your own implementation. As for the providers for the tree viewer, we recommend creating an own implementation of both, the `IContentProvider` and `IBaseLabelProvider`, interfaces and return the data in an appropriate form for the requirements in your application. Like this, you can specify yourself how the data should be returned

and presented to the user.

Model data

To set the input data in the outline page we recommend creating a public method `setModel` in your outline page implementation, which sets the input on the `TreeViewer`. The call of this method should be coupled with the reconciler to ensure the model is up-to-date if the content in the text editor document changes.

Alternatives

As you can see, the RCP framework provides a good basis for the outline page implementation and more or less gives a border how this should be done. Surely, there are alternative solutions possible, but we do not recommend creating your own implementation for the whole outline view and outline page handling, if the requirements of your application do not request it especially.

8.3.9.4 Content Assistance

Overview

By giving the user contextual content proposals, typing effort and typing errors can be reduced.

Point of extension

The implementation of the content assistance is done through the extension of the `SourceViewerConfiguration` by overriding the method `getContentAssistant` and by returning an `IContentAssistant`.

Concept

The RCP framework provides a default implementation of `IContentAssistant` called `ContentAssistant`, which we recommend to use for the implementation of the `getContentAssistant` method in the `SourceViewerConfiguration`.

Implementation

A `ContentAssistant` uses an `IContentAssistProcessor` to compute the content proposals for each content type in the document. According to the RCP framework documentation, it is intended that the client creates its own implementation of `IContentAssistProcessor`. Therefore, no default implementations are provided. That's why we also recommend creating your own implementation and returning appropriate completion proposals based on the requirements of your application.

The example in **Listing 197** shows how the content assistance can be configured and integrated.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {
  ContentAssistant assistant = new ContentAssistant();

  IContentAssistProcessor processor =
      new PersonLocatorCompletionProcessor(fEditor);
  assistant.setContentAssistProcessor(processor,
      IDocument.DEFAULT_CONTENT_TYPE);

  assistant.setContextInformationPopupOrientation(
      IContentAssistant.CONTEXT_INFO_ABOVE
  );
```

```
  assistant.setInformationControlCreator(
      getInformationControlCreator(sourceViewer)
  );
  assistant.enableAutoInsert(true);

  return assistant;
}
```

Listing 197: Example showing the implementation content assistance

Action

In order to make it possible for the user to call the content proposals by key-press, an `IAction` has to be defined in the text editor implementation, like shown in **Listing 198** below. We recommend using a `TextOperationAction`, because it is an action associated with the text editor and intended by the framework to be used for such cases.

```
protected void createActions() {
  super.createActions();

  IAction action =
    new TextOperationAction(
    PersonLocatorEditorPlugin.getDefault().getResourceBundle(),
          "ContentAssistProposal.",
          this, ISourceViewer.CONTENTASSIST_PROPOSALS
    );

  action.setActionDefinitionId(
      ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS
  );

  setAction("ContentAssist.", action);
  markAsStateDependentAction("ContentAssist.", true);
}
```

Listing 198: Example showing the creation of the actions in the text editor

The operation actions are intended to be configured through a property file using a `ResourceBundle`. Therefore, we recommend creating a property file with your custom configuration in your project and providing a `getResourceBundle` method in your `AbstractUIPlugin` implementation, which returns the `ResourceBundle`.

Action bar integration

Additionally, the action has to be assigned to the content assistant in the contributor class. To do this, we recommend using a `RetargetTextEditorAction`, which can be retargeted to dynamically changing actions from the active editor. The **Listing 199** below shows an example implementation of such an integration.

```
public class PersonLocatorEditorContributor
        extends BasicTextEditorActionContributor {

  ...

  private RetargetTextEditorAction fContentAssist;

  public PersonLocatorEditorContributor() {
    super();
```

```
    fContentAssist =
        new RetargetTextEditorAction(
            PersonLocatorEditorPlugin.
                getDefault().getResourceBundle(),
            "ContentAssistProposal."
        );
    fContentAssist.setActionDefinitionId(
        ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS);
  }

  public void setActiveEditor(IEditorPart part) {
    super.setActiveEditor(part);
    ITextEditor editor =
       (part instanceof ITextEditor) ? (ITextEditor) part : null;

    fContentAssist.setAction(
        getAction(editor, CONTENTASSIST_ACTION)
    );
  }

  public void init(IActionBars bars, IWorkbenchPage page) {
    super.init(bars, page);
    bars.setGlobalActionHandler(
        CONTENTASSIST_ACTION, fContentAssist
    );
  }

  ...
}
```

Listing 199: Example showing the implementation of the contributor class

Alternatives

In this case, too, the framework gives fairly strict boundaries how the content assistance should be implemented. Therefore, there is not a lot of space available for alternative implementations.

By creating an own `IContentAssistant` implementation, specific content assistants with custom behavior can be created. Because of the high implementation effort and the great look and feel of the provided default implementation, we only recommend implementing an own content assistant in cases where the default implementation does not cover the requirements of your application.

8.3.9.5 Annotations

Overview

Annotations are pieces of information for specific positions in a text. By providing extra information on text positions of a document, the understanding of its meaning and its structure can be supported.

Point of extension

Besides the `IDocument` model, which is used to access the text of a document, the RCP text facility, more precisely the `ISourceViewer`, provides an `IAnnotationModel`, which allows the inclusion of extra information on text positions.

Typically, the annotation model is associated with changes in the application lifecycle. Because the `AbstractDecoratedTextEditor` provides access to the `ISourceViewer` and carries different change listeners, we recommend using the text editor as the point of extension for the introduction of annotations.

Recommendation

For the addition of annotations to the `IAnnotationModel`, the `ISourceViewer` is required. The annotation model can be retrieved from the source viewer and allows the addition of annotations through the method `addAnnotation`. To be able to show the annotations in the text editor, the framework requires additionally a description of the position in the text. The `Position` class provides this functionality and can be used for this purpose.

Taking an occurrences updater as an example, an `ISelectionChangedListener` can be implemented and added to the text editor. It allows reacting on selection changes and thus recognizing where respectively on which word the cursor stands. Like this, annotations can be created for all existing words in the document. The best place to implement such functionality is, as already mentioned before, in the text editor implementation class. Here, one can react on model changes, like e.g. when the model reconciliator is invoked, and has all the information required for the proper handling.

Alternatives

The Eclipse framework specifies the implementation of the annotations quite strictly. Therefore there are no great possibilities for custom implementations and thus there is no space for large alternatives.

8.3.9.6 Context Menu

Overview

Providing additional functionality in the context menu can help make the application more practical and useful for the user. Especially in cases where the content and the selection in the editor are relevant to create domain objects for further use in the application.

Concept

For the implementation of additional functionality in the context menu of the text editor we recommend using the Eclipse command framework. The command framework is a newer concept which decouples the definition of a command from its handling and thus increases the flexibility and reusability of the application.

Extension point

Using the extension point `org.eclipse.ui.commands`, the commands for the different operations can be defined. We recommend categorizing the commands based on their main concern by creating a command category (`category`). As for the icons of the operations, we recommend using the extension point `org.eclipse.ui.commandImages`. By linking the images (`image`) with commands they appear wherever the command is used in the application.

Now that the commands are defined, they can be placed in designated places in the application. For the positioning of the commands we recommend using the extension point `org.eclipse.ui.menus`. By defining a `menuContribution` with the `locationURI` *"popup:org.eclipse.ui.popup.any"* a contribution to all registered context menus in the application can be realized. The `menuContribution` clause contains references to the commands, which will be provided in the specified location.

Command handling

For the handling of the commands we recommend creating command handlers by using the extension point `org.eclipse.ui.handlers`. A handler (`handler`) has a link to the command and a reference to a handler implementation class. Handler implementations should implement the interface `IHandler` in

order to be recognized as handlers in the framework.

A more detailed description on this topic is available in chapter 8.3.5.

Dynamic entries

In the case of the text editor, the `BasicTextEditorContributor` class of the Eclipse framework is used. It provides a method called `contributeToMenu` which has to be overridden in order to contribute actions to the context menu. In this method, custom actions (`IAction`) can be added to the `IMenuManager`. Like this, it is possible to create dynamic actions and thus context menu entries based on the current application state.

A more detailed description about dynamic actions can be found in chapter 8.3.5.

8.3.9.7 Hints

Studying the framework

Considering the lack of good documentation about the RCP framework, the best way to find an introduction into this area is to read the Javadoc description and study the default implementations provided in the framework.

As for the text editor implementation, having a look into the Javadoc and the implementation of the following classes can be helpful:

- Common text facility interfaces and classes
  - ITextEditor
  - AbstractDecoratedTextEditor
  - IEditorActionBarContributor
  - SourceViewerConfiguraiton
- Syntax highlighting
  - IPresentationReconciler
  - PresentationReconciler
  - IPresentationDamager, IPresentationRepairer
  - ITokenScanner
  - RuleBasedScanner
  - IRule
- Outline page
  - IContentOutlinePage
  - ContentOutlinePage
  - ITreeContentProvider
  - LabelProvider
- Model reconciliation
  - IReconciler
  - MonoReconciler, Reconciler
  - IReconcilingStrategy, IReconcilingStrategyExtension

- Content assistance
  - IContentAssistant
  - ContentAssistant
  - IContentAssistProcessor
  - IAction
  - TextOperationAction
  - RetargetTextEditorAction
- Annotations
  - IDocument
  - IAnnotationModel

8.3.10 Undo/Redo

Overview

To undo and redo operations in a graphical user interface is a core requirement of today's applications and are expected by the users as a standard feature. For this reason, such a handling is included by default in Eclipse RCP, which provides common interfaces and abstract classes for its implementation. This covers typically the command objects, because the whole feature is based on the *Command Pattern [GoF1995]* respectively its extension the *Command Processor [Buschmann1996]*.

In this chapter, we take a detailed look at this feature. We show you the elementary classes and interfaces and how you can extend them with your specific implementations for your own RCP based application.

8.3.10.1 Structure

Description

Although Eclipse RCP simplifies the implementation of the undo/redo functionality, it requires a clear design from the very beginning. The reason for this requirement is the structure of this feature. It consists of several components which work closely together.

Diagram

The structure of the operation processing consists of the components and parts shown in **Figure 47**.

Triggered through a user action, an operation, also called command object, will be created. It contains the specific operation steps, which should be applied on the target objects. It will be executed as a transactional object. Additionally, it can, but must not, be embedded in a Job. This allows a decoupling of the execution context through an asynchronously invocation. Finally, they will be put on the command stack of the platform, which holds them for undo purposes.

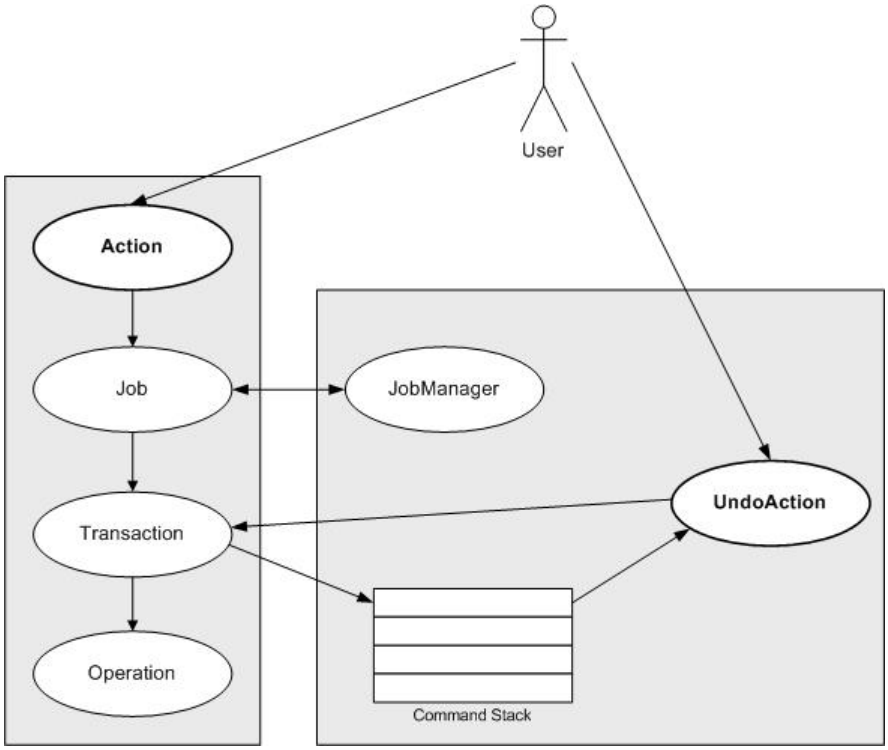

Figure 47: Operation processing structure in RCP

Please note the use of *Jobs* in combination with a *JobManager* for the asynchronous execution of the transactional objects is optional. The decision to use it or not is up to you.

8.3.10.2 IUndoableOperation

General

An undo and a redo of a specific operation, like a text formatting, suppose a description of it or something else. Otherwise, the application doesn't know what should be undone and in what way.

The Gang of Four *[GoF1995]* suggested with the *Command Pattern* an easy and very flexible solution to solve this problem. The key characteristic of the proposed solution is the encapsulation of the operations in own classes implementing a common interface, like mentioned above.

The developers of Eclipse RCP, to which Erich Gamma from the Gang of Four belongs, take the idea of the *Command Pattern* and extended it to provide a powerful, flexible and easy solution.

8.3.10.2.1 Interfaces & Classes

IUndoableOperation

The common interface for this purpose is the `IUndoableOperation`. It defines beside the obvious methods for the execution, undo and redo invocation, also methods to control its execution. An example of such configuration methods are `canExecute` or `canUndo`. They allow separating the command execution from the checks if it can be executed. This helps to keep your implementation clean

and allows the framework to adjust their behavior on the availability of the command execution.

For this reason, we recommend you to implement your commands with this separation. Otherwise, you have to implement the check in the execution methods and notify the user by your own about command steps, which cannot be executed, undone or redone.

The methods for executing, undoing and redoing have all the same parameter list. The first parameter they get is an instance of `IProgressMonitor`, which allows you to interact with a progress monitor to show the working progress. The second parameter is an instance of `IAdaptable` which is provided by the caller in order to provide required resources in a generic way. You can use it for example to prompt the user if necessary.

They all have also the same return value type, which must be an `IStatus` object. The class `Status` provides a default implementation of the `IStatus` interface you can use. If the method's execution completed correctly we recommend you to return a state with `IStatus.OK`, but if an error occurred the return value `IStatus.ERROR` is appropriate.

AbstractOperation

With the abstract class `AbstractOperation`, RCP provides a default implementation of the `IUndoableOperation` interface. If you use this class for your own command objects, the inheritance forces you only to override the `execution`, `undo` and `redo` methods. Further methods can be overridden if necessary.

Because of this simplification of the command object implementation, we recommend you to use the abstract class instead of the interface. Otherwise, you have to implement every time all method, even if you don't need them.

Example

Now we show you a simple example in **Listing 200** to get an impression how such operation looks like. The example adds two numbers and passes the result afterwards to a model class.

```
public class AddCalculationCommand extends AbstractOperation {

  private CalculatorModel model = null;
  ...

  public AddCalculationCommand(CalculatorModel model) {
    super("Add");
    this.model = model;
  }

  public void setOperand1(int operand1) {
    this.operand1 = operand1;
  }

  public void setOperand2(int operand2) {
    this.operand2 = operand2;
  }

  @Override
  public IStatus execute(IProgressMonitor monitor, IAdaptable in-
fo)
    throws ExecutionException {
```

```
    int result = operand1 + operand2;
    model.setValue(result);
    return Status.OK_STATUS;
  }

  @Override
  public IStatus redo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException {

    return execute(monitor, info);
  }

  @Override
  public IStatus undo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException {

    model.setValue(oldValue);
    return Status.OK_STATUS;
  }
}
```

Listing 200: Example implementation of an IUndoableOperation

8.3.10.3 Execution of operations

General

The execution of the command objects is more than just invoking the execute methods. It implies also a mechanism to cache the executed operations. Otherwise, they would not be available for an undo and redo in your application.

In Pattern-Oriented Software Architecture *[Buschmann1996]* the authors suggested with the *Command Processor* a flexible and powerful extension to the *Command Pattern*. Beside the command specification, this pattern defines also how to execute and manage the commands. This is especially for the undo and redo functionalities an important fact.

Implementation

Based on this idea, Eclipse RCP provides also an extended implementation of a command processor which can be used and configured in detail.

We recommend you to implement the execution of the command objects in a general manner. The reason for this recommendation is to implement it only once, which allows a reduction of code. This leads to a further recommendation to locate this implementation in a utility or core bundle, for an easy usage in your whole application. Otherwise, you have the risk of duplicate implementations of the execution statements in several bundles, which lead to duplicate code. Exactly this should be avoided.

Execution

The execution of command objects itself is based on a solution using two objects, one for the operation support and the other for the execution and the history keeping. You can get them from the workbench of your RCP application like shown below.

```
IWorkbench wb = PlatformUI.getWorkbench();
```

Operation support

From the workbench you can get the first object which implements the `IWorkbenchOperationSupport` interface. It has two main purposes. You can use it to get the operation history, the second object you need for the command object execution, and you can configure your operations with the provided undo con-

text.

Operation history

The second object implementing the `IOperationHistory` interface is needed for the execution of your command objects using the provided `execute` method.

The use of these two classes for the execution of the command objects is mandatory. Therefore we cannot give you some other recommendations than following the suggestions of the RCP developers.

Example

The following code snippet in **Listing 201** below shows a method for the execution of the command objects.

```
public IStatus performOperation(IUndoableOperation op,
  IProgressMonitor monitor, IAdaptable info) {

  IWorkbench workbench = PlatformUI.getWorkbench();
  IWorkbenchOperationSupport operationSupport
                       = workbench.getOperationSupport();
  IOperationHistory operationHistory
                       = operationSupport.getOperationHistory();

  // Add the operation to the UndoContext
  op.addContext(operationSupport.getUndoContext());
  // Set the maximum number of undo steps
  operationHistory.setLimit(
      operationSupport.getUndoContext(), 10);

  try {
    // Execute the operation and add it to the history
    return operationHistory.execute(op, monitor, info);
  } catch (ExecutionException e) {
    return new Status(IStatus.ERROR, PLUGIN_ID, NLS.bind(
      "Cannot execute operation {0}", op.getLabel()), e);
  }
}
```

Listing 201: Example implementation of the command object execution

Customization

Like you've already seen in the example from **Listing 201** above, the execution of command objects can be configured. This covers especially setting the undo context and a limit of the corresponding history size, which must be a positive value.

With the undo context you can control the grouping of the command objects. This allows you the filtering of the operation history for the undo and redo visibility, in such a way that only the ones for a given context are displayed.

We recommend you to use in smaller applications the global undo context. Everything else would be a huge overhead without advantage.

Different are larger applications. We recommend you to use in these cases different undo contexts depending on the different views of the user interface. It improves the undo/redo handling, because only the undoes and redoes possible for the active view are displayed. As an additional advantage you can configure a lower operation history limit, because different histories must contain fewer entries.

Of course, you can configure your command objects with further options, but we don't want to explain them here in detail. You can get a detailed description of them in the RCP Javadoc if you need them.

Example

In **Listing 202** we want you to show an example of an `IUndoContext`, which allows you to return a label and to check if the passed context matches with the own one.

```
public class MyUndoContext {

  public boolean matches(IUndoContext context) {
    return context == this;
  }

  public String getLabel() {
    return "My undo context";
  }
};
```

Listing 202: Example implementation of an IUndoContext

8.3.10.4 Use Jobs and the JobManager for operations

General

With the encapsulation of the user interactions in command objects and its execution described above it is not possible to coordinate it with the execution of your commands. For this reason the Eclipse RCP provides Jobs and the *JobManager*. Besides the temporal decoupling of the execution from the creation of the command objects, its big advantage is the integrated concurrency coordination. This allows you to get exclusive access to resources easily.

Implementation

Generally we recommend you to implement your jobs in a general manner and to locate them in central bundle, like in a utility. The reason for those recommendations is mainly the reduction of duplicate code, because otherwise you would have the risk to implement specific jobs for different purposes, which are very similar.

Additionally you can keep your dependencies low through the location in a utility bundle, which is typically used by the most other bundles of your application. Otherwise you would add further dependencies which can lead to cycles.

The job

For the implementation of the job classes itself RCP provides the abstract class Job. It forces you to override the method `run` and to implement a constructor passing the name of the job.

We recommend you to define the constructor as private and to provide factory methods instead, like shown in **Listing 203**. The advantage of such an implementation is in our opinion, because it allows beside the creation of a new job instances also its configuration. Otherwise they must be done in the invoker several times.

Configuration

Possible configurations are the execution priority or the scheduling rule to allow an optimal command execution. The main purpose of the scheduling rule is described by the Javadoc as follows: *"The scheduling rules are used by the jobs*

to indicate when they need exclusing access to a resource. The job manager guarantees that no two jobs with conflicting scheduling rules will run concurrently."

So we recommend you to define them for the different resources and to set them if your command requires exclusive access on them. Otherwise you have to take care on your own for a proper synchronization, which is cumbersome and error-prone.

Command execution

Beside the creation and the scheduling of the job you also have to implement the execution of the passed command object. The run method is the appropriate location for this implementation, because it is invoked from the JobManager after the scheduling of the job.

We recommend you to invoke the class in the utilities, which contains the configuration of the operation support and the command execution. It is also possible to do such an implementation in this method, but we disadvise you to do it. The reason is on the one hand the coupling of this class, which would be increased unnecessarily, and on the other hand because these implementations have to be done several times, which leads again to duplicate code.

The following **Listing 203** shows an example of such a job implementation.

```
public class OperationJob extends Job {

  private static ISchedulingRule rule = new ISchedulingRule() {
    public boolean contains(ISchedulingRule rule) {
      return rule == this;
    }

    public boolean isConflicting(ISchedulingRule rule) {
      return rule == this;
    }
  };

  public static OperationJob executeOperation(
        IUndoableOperation operation, IAdaptable info) {
    OperationJob job = new OperationJob(operation, info, false);

    job.setPriority(INTERACTIVE);
    job.setRule(rule);
    job.schedule();
    return job;
  }

  ...

  private IUndoableOperation operation;
  private IAdaptable info;
  private boolean joinSwtThread;

  private OperationJob(IUndoableOperation operation,
        IAdaptable info, boolean joinSwtThread) {
    super(operation.getLabel());
    this.operation = operation;
    this.joinSwtThread = joinSwtThread;
    this.info = info;
  }

  @Override
```

```
  protected IStatus run(final IProgressMonitor monitor) {
    IStatus status;

    if (joinSwtThread) {
      Shell shell = (Shell) info.getAdapter(Shell.class);
      if (shell != null) {
        shell.getDisplay().syncExec(new Runnable() {
          public void run() {
            status = new OperationUtil().performOperation(
                operation, monitor, info);
          }
        });
        return status;
      }
    }

    return new OperationUtil().performOperation(
        operation, monitor, info);
  }
}
```

Listing 203: Example of a job implementation for the asynchronously execution of commands

8.3.10.5 User interface

General

After the implementation of the undo and redo functionality in the core of your application, you must facilitate the user to undo and to redo these operations. The RCP provides some simplifications for the integration of the undo and redo functionality in the UI of your application.

Action bars

A first possibility provides the action bars, which includes amongst others the menu, tool and cool bar. You don't have to create own commands and handlers, because RCP already provides them under the *commandId* `org.eclipse.ui.edit.undo` and `org.eclipse.ui.edit.redo`. How you can extend them with further commands is described in chapter 8.3.5.

We recommend you to add the undo and redo entries to the action bars, because they are expected from the users. Additionally they allow the usage of the entries through the key assist.

Buttons, etc.

Further possibilities provide the implementation of own solutions using UI elements, like buttons. Compared with the action bars their use is less intuitive and more complex in the development, because the operation history must be notified programmatically in the same way like for the command execution (see chapter 8.3.10.3). So we don't recommend such an implementation.

Link with your view

Using the global undo context is typically enough for the user interface integration. But in some cases the undo and redo entries won't be activated or display the wrong undo and redo commands. In this case the link of your view with the actions is missing or wrong.

To add the link for the undo and redo in your view, you need the undo context used for the command execution. Afterwards you have to create instances of the `UndoActionHandler` and `RedoActionHandler` using the undo context. They represent the classes behind the action bar entries, which must be regis-

tered on the action bar. For this purpose you can use the `ActionFactory` with the id of the commands.

Example

In **Listing 204** below such an implementation linking the view with the global action bars is shown using the global undo context.

```
// Undo/Redo
private IUndoContext undoContext
    = IOperationHistory.GLOBAL_UNDO_CONTEXT;
private UndoActionHandler undoAction;
private RedoActionHandler redoAction;
```

```
// Link with the Undo/Redo Actions
undoAction = new UndoActionHandler(this.getSite(), undoContext);
redoAction = new RedoActionHandler(this.getSite(), undoContext);

IActionBars actionBars = getViewSite().getActionBars();

actionBars.setGlobalActionHandler(
    ActionFactory.UNDO.getId(), undoAction);
actionBars.setGlobalActionHandler(
    ActionFactory.REDO.getId(), redoAction);
```

Listing 204: Example implementation of undo/redo linkage in the view

8.3.11 Common Navigator Framework

Overview

The *Common Navigator Framework* is a compact framework for the implementation of application internal navigators. Depending on the purpose of an application such a navigator can be used for the management and organization of files in projects. Thereby they can be used as objects in the application, which allows completely new opportunities.

As a part of the bigger Eclipse Rich Client Platform it allows a seamless integration with the remaining parts of you RCP application, for example with the different editors.

8.3.11.1 Implementation

Its implementation consists of several parts, which includes mainly the view and its configurations.

View

The view itself is very simple to develop, because it already exists with the id `org.eclipse.ui.navigator.ProjectExplorer`. Therefore we recommend you to use this view instead of defining an own one. Otherwise you have to do also the default configurations on your own, which would be duplicated, one time in the Eclipse plug-in and a further time in your own one.

Its addition to a perspective would looks like shown in the following **Listing 205**.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.personlocator.ui.perspective">
    ...
    <view
      closeable="true"
      id="org.eclipse.ui.navigator.ProjectExplorer"
      minimized="false"
      moveable="true"
      ratio="0.25"
      relationship="left"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      standalone="true"
      visible="true">
    </view>
    <viewShortcut
      id="org.eclipse.ui.navigator.ProjectExplorer">
    </viewShortcut>
    ...
  </perspectiveExtension>
</extension>
```

Listing 205: Addition of the `org.eclipse.ui.navigator.ProjectExplorer` to a perspective

Own view

But if you want to define an own navigator view, we recommend you to use the extension point `org.eclipse.ui.views`, like shown in **Listing 206** below, because you can define the whole view without writing a single line of code. This allows a very flexible definition of the navigator without dependencies to the concrete classes. Additionally you have the possibility to configure the navigator using extension points with the same advantages compared with the programmatic way. Also the addition of this view to perspectives through extension points is possible.

Otherwise you have to create a new class inheriting from `CommonNavigator`. It works fine but is an overhead, compared with the declarative definition. Additionally you have to do the addition of the view to perspectives programmatically as well as the further configurations. But such a solution is not very flexible why we disadvise you to do it in this way.

So generally you have a more flexible solution with fewer dependencies to RCP classes using the extension points.

```
<extension point="org.eclipse.ui.views">
  <view
    class="org.eclipse.ui.navigator.CommonNavigator"
    id="org.herasaf.tutorial.personlocator.ui.Navigator"
    name="Navigator"
    restorable="true">
  </view>
</extension>
```

Listing 206: Definition of the navigator using the `org.eclipse.ui.views` extension point

8.3.11.2 Configuration

Configuration

After the definition of your navigator you have to configure it for its use. General-

ly we recommend you to do the configurations in the same way you also defined the navigator. If you defined it through extension points, we recommend you to do its configuration in the same way. Otherwise you have the risk of incompatibilities between the two solutions or it is not possible to do in another way.

Default configurations

If you are use the view with the id `org.eclipse.ui.navigator.ProjectExplorer`, the default configurations are already done. So you have to do only the ones specific for your application.

This is different if you defined an own view for the navigator. Here have to do in a first step the default configurations, because they provide the navigator's base functionality. Based on them you can afterwards do the specific configurations. Otherwise you have the risk your extensions won't work as expected because they require some basic functionalities which are already implemented by the default configurations.

So for this reason we recommend you to add the default configurations to you navigator if you'd defined it in an own view. You can do this using the extension point `org.eclipse ui.navigator.viewer` or programmatically in the class inheriting from `CommonNavigator`.

Using extension points

Using the extension point for setting the default configuration covers the three parts, like show in **Listing 207**. The first one is telling which view part is a Common Navigator using the `viewer` tag. Here you have to write the id of the view you'd defined previously.

The second part of the configuration is setting the action bindings. Here also the id of the affected view must be added as well as the pattern of the default actions' ids like propose in *[CNFProgGuide]*. You can also add each id of the action provider on your own, but it's a bit complicated.

The last part is setting the viewer content, which defines the possible resource, filter, link helper and working set support.

```
<extension point="org.eclipse.ui.navigator.viewer">
  <viewer
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
  </viewer>
  <viewerActionBinding
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
    <includes>
      <actionExtension
        pattern="org.eclipse.ui.navigator.resources.*">
      </actionExtension>
    </includes>
  </viewerActionBinding>
  <viewerContentBinding
    viewerId="org.herasaf.tutorial.personlocator.ui.Navigator">
    <includes>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resourceContent">
      </contentExtension>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.filters.*">
      </contentExtension>
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.linkHelper">
      </contentExtension>
```

```
      <contentExtension
        pattern="org.eclipse.ui.navigator.resources.workingSets">
      </contentExtension>
    </includes>
  </viewerContentBinding>
</extension>
```

Listing 207: Configuration of the navigator using the extension point `org.eclipse.ui.navigator.viewer`

Programmatical

Different are the default configurations if you do it programmatically. Here you have to override the methods `getNavigatorActionService` and `getNavigatorContentService` in your navigator class. In these methods you have to add the object instances of the classes, which are used in the extension points to define the default configuration.

But this is very cumbersome and needs a huge effort. Therefore we disadvise you to do the configurations in a programmatically manner.

8.3.11.3 Extension

Extensions

Based on the default configurations you can build your own actions, viewer contents, popup menus and further more through the simple extension of the existing parts. For these specific configurations the same applies as for the default ones already mentioned in the chapter above. We recommend you to do it in the same way you also defined your navigator.

Eclipse RCP provides for this purpose the extension point `org.eclipse.ui.navigator.navigatorContent`. Here you can define navigator content providers, further action providers, filters and further more. *[Elder06]* explains such an individual extension for a simple properties file. So we refer to its descriptions and explanations.

8.3.11.4 Hints

Ids of the default configurations

For doing the default configuration of your navigator you can use the already provided base functionalities. You can find the ids and classes of the action providers, viewer contents and further parts for the declarative and programmatically definition in the *plugin.xml* file of the plug-in *org.eclipse.ui.navigator. resources*.

8.3.12 Problems View

Overview

The *Problems View* component allows showing the current errors, problems and warnings in files, graphical diagrams and further more in a detailed and clear manner. Therefore it gives the user an overview about the current state.

Compared with notifications given by popups and dialogs, which are only shown as long as the user closes it, the *Problems View* shows them for a longer period

but without hinder the user. This makes it a very useful feature.

8.3.12.1 Implementation

The implementation of the *Problems View* is very compact, because several parts are already implemented and only have to be configured using extension points or inheritance from existing classes.

Generally we recommend you to do the view definition and configuration using extension points. Beside its flexibility and reduced coupling it needs fewer effort compared with the implementation way. Additionally it's hard to find descriptions how to do the configurations programmatically.

View

The view itself must not be defined, because Eclipse RCP provides it already with the id `org.eclipse.ui.views.ProblemView`. Therefore you can add it simply to your perspective like show in **Listing 208** below.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.herasaf.tutorial.personlocator.ui.perspective">
    ...
    <view
      closeable="true"
      id="org.eclipse.ui.views.ProblemView"
      minimized="false"
      moveable="true"
      ratio="0.66f"
      relationship="bottom"
      relative="org.eclipse.ui.editorss"
      showTitle="true"
      visible="true">
    </view>
    ...
  </perspectiveExtension>
</extension>
```

Listing 208: Code excerpt showing the addition of the *Problems View* to the perspective

Marker definition

But when you start your application now, you will see the table has not columns. So it is required to configure them. The easiest way is also here the declarative configuration. Eclipse RCP provides for this purpose the extension point `org.eclipse.ui.ide.markerSupport`.

As sub elements you can define columns (`markerField`), configure the *Problems View* itself (`markerContentGenerator`), create grouped columns (`markerGrouping`) and further more. We recommend you to use mainly the already existing fields, because wide amount of them is already provided. Otherwise you have a huge effort to define and implement them.

So the only thing you have to do typically is the composition of the *Problems View*. For this purpose you have to create a `markerContentGenerator` with the id `org.eclipse.ui.ide.problemsGenerator` and add the fields of your choice. The **Listing 209** shows an example of such a configuration.

```
<extension point="org.eclipse.ui.ide.markerSupport">
  <markerContentGenerator
    id="org.eclipse.ui.ide.problemsGenerator"
    name="Problems Generator">
    <markerFieldReference
      id="org.eclipse.ui.ide.severityAndDescriptionField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.resourceField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.pathField">
    </markerFieldReference>
    <markerFieldReference
      id="org.eclipse.ui.ide.locationField">
    </markerFieldReference>
    <markerTypeReference
      id="org.eclipse.core.resources.problemmarker">
    </markerTypeReference>
  </markerContentGenerator>
</extension>
```

Listing 209: Example configuration of the *Problems View* with markers using the extension point `org.eclipse.ui.ide.markerSupport`

8.3.12.2 Editor integration

Introduction

So the steps we did until now were simply to define and configure the visualization of the *Problems View*. But now it is required to fill it with errors, problems, warnings and further more.

This can be done based on `IResource` objects. This means for example files, folders and other resources. Consequently the obviously components for this purpose are the editors.

Structure

But before we start to implement the editor integration, we want to give you a general advice. We recommend you implement the creation and update of the markers, like the entries in the *Problems View* are called, in an own class with a common interface, like shown in **Listing 210** below.

```
public interface IMarkerHandler {

  public void updateMarkers(ITextEditor editor,
      List<ParseException> parseExceptions);
}
```

Listing 210: Example of a common interface for the marker creation and update

Through such a solution you have the possibility for an easy replacement of an implementation through substitutes without changes in the editor class itself. This is an interesting feature, especially if the factories or Spring's dependency injection are used. Additionally a concrete implementation can be used several times, which reduces duplicate code.

Marker handler

To create and update existing markers you have to get the `IResource` object in a first step. Typically such an object is contained in or reachable over the editor input.

In a further step we recommend you to delete all existing markers to avoid duplicate or entries which are not longer up-to-date. You can get all of them from the `IResource` through its `findMarkers` method. Afterwards you can invoke on each object its `delete` method.

Finally you can create the new and up-to-date markers through the `createMarker` method of the `IResource` object. The obtained object can be configured with attributes like the severity, a message and further more.

All remaining steps, for example to update the *Problems View*, are handled by Eclipse RCP.

The **Listing 211** below shows such an example implementation for text editors.

```
public class DefaultMarkerHandler implements IMarkerHandler {

  protected void clearMarkers(IFile file) {
    try {
      IMarker[] markers = file.findMarkers(IMarker.PROBLEM, true,
          IResource.DEPTH_INFINITE);
      for (IMarker marker : markers) {
        marker.delete();
      }
    } catch (CoreException e) {
      e.printStackTrace();
    }
  }

  private int getDocumentPosition(ITextEditor editor,
      int lineNumber, int position) {

    int documentPosition = -1;

    try {
      IDocument document = editor.getDocumentProvider().
          getDocument(editor.getEditorInput());
      documentPosition = document.getLineOffset(lineNumber);
      documentPosition += position;
    } catch (BadLocationException e) {
      // do nothing
    }

    return documentPosition;
  }

  public void updateMarkers(ITextEditor editor,
      List<ParseException> parseExceptions) {

    if (editor.getEditorInput() instanceof IFileEditorInput) {
      IFileEditorInput input =
          (IFileEditorInput) editor.getEditorInput();
      IFile file = input.getFile();

      clearMarkers(file);

      for (ParseException parseExc : parseExceptions) {
        try {
          IMarker marker = file.createMarker(IMarker.PROBLEM);
```

```
            marker.setAttribute(IMarker.SEVERITY,
                IMarker.SEVERITY_ERROR);
            marker.setAttribute(IMarker.MESSAGE,
                parseExc.getMessage());
            marker.setAttribute(IMarker.LINE_NUMBER, parseExc
                .getLineNumber());
            marker.setAttribute(IMarker.CHAR_START,
                getDocumentPosition(editor, parseExc
                    .getLineNumber(), parseExc
                    .getStartPosition()));
            marker.setAttribute(IMarker.CHAR_END,
                getDocumentPosition(
                    editor, parseExc.getLineNumber(), parseExc
                    .getEndPosition()));
          } catch (CoreException e) {
            e.printStackTrace();
          }
        }
      }
    }
}
```

Listing 211: Example implementation of a marker handler for text editors

Editor

In the editor class itself you only have an instance of an object with the common interface. After your parser, compiler or other component has checked the editor's content the marker handler must be invoked to update the *Problems View*. That's all what you have to do for the editor integration itself.

8.3.12.3 Hints

Problems View update in the SWT thread

We recommend you to invoke the update of the markers for the *Problems View* in the SWT thread to prevent problems with the synchronization. Otherwise you have the risk to get exceptions caused by accessing the user interface from a different a non-authorized thread.

```
Shell shell = getSite().getShell();
shell.getDisplay().asyncExec(new Runnable() {
  public void run() {
    markerHandler.updateMarkers(
        «Class».this, parseExceptions);
  }
});
```

Listing 212: Code snippet showing the marker handler invocation in the SWT thread

8.3.13 Multilanguage support

Overview

The Eclipse framework by default provides a string externalization mechanism, which supports the developer in finding and managing loose strings in the application. In this chapter we give recommendations about how you can extend your application to provide multilanguage support.

Implementation

For the implementation of the multilanguage support, we recommend using Eclipse's string externalization mechanism. It helps finding loose strings, generates the required utility class and fills up the messages property file automatically.

In order to avoid changes in multiple projects and not to lose the overview we recommend creating a central bundle which holds all the messages for the application. Another solution would be to place the messages in the specific bundle itself. In a large application, like e.g. in the Eclipse IDE, it can be reasonable to choose such a decentral solution. Which solution to choose, really depends on the requirements of the application to be developed. Is it intended to provide a core for further extensions and to be used as a framework, the decentral solution is more appropriate. Otherwise, in an application which is implemented and enhanced as a whole, a central solution avoiding the search where changes have to be done can be reasonable, too.

Messages can be accessed through the generated class `Messages` by using the method `getString`.

Addition of further languages

Concerning the addition of further languages we recommend extending the language bundle with fragment bundles. The fragment bundle holds the messages property file with the appropriate translations for the specific language and makes them available when it is installed in the framework.

8.3.14 Wiring with Spring

Overview

The use of Spring also in a RCP application is very useful. Through its dependency injection you can gain the flexibility in your implementation to implement fully against interfaces and abstract classes. You don't have to consider which concrete implementation is used as long as the contract is fulfilled.

Spring Framework Utilities

But the use of Spring in a RCP application is very limited through the extension point concept and other implementation details. But with the *Spring Framework Utilities* invented by Martin Lippert *[Lippert08]* you have now the possibility to intervene directly in the extension points. This means you can use in the extension point classes instantiated and managed by Spring.

You can achieve this through the replacement of the concrete class in the extension point with the `org.eclipse.springframework.util.SpringExtensionFactory`, followed by a colon and the name of the bean out of the application context. The factory looks up the bean with the passed name and returns it. The extension point itself doesn't notice anything.

We recommend you to use this simple Spring extension. Otherwise you have to implement it on your own to get the beans out of the application context, which is cumbersome and increases the dependencies unnecessarily. Therefore real alternatives don't exist.

8.3.14.1 Hints

Inject OSGi services

Please consider well the injection of OSGi services in your classes, especially if they are mandatory. It is possible that your application blocks, because the re-

quired services isn't available at the moment you wire your component.

You can solve such problems in two ways. The first one is to configure the start order of the bundles on your own. So you can control which bundle must be started before another. It works well, whereas it breaks the idea of the OSGi framework, where bundles can be started and stopped at any time.

The other and in our opinion best solution is to change the cardinality to *0..1* respectively *0..N*. But it has the disadvantage you must implement an appropriate handling if the service is not available. This can be a cache for data which must be saved or something else.

Constructor parameters

In some cases you will need to configure an RCP component with a class, where resources of the invoker must be passed in the constructor. In these cases a wiring is not possible, because Spring has no access on the required resources.

To allow a wiring with Spring nevertheless, we recommend you to implement an indirection layer, whose purpose is to instantiate the required class with the passed parameters, like shown in **Listing 213** below.

```
public interface ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
        EditPartViewer viewer, ActionRegistry registry);
}


public class DefaultContextMenuProviderFactory implements
      ContextMenuProviderFactory {

  public ContextMenuProvider createContextMenuProvider(
        EditPartViewer viewer, ActionRegistry registry) {
    return new DefaultContextMenuProvider(viewer, registry);
  }
}
```

Listing 213: Example of a indirection layer to allow a proper Spring wiring

8.4 Control Layer

Intention

According to the advantages a control layer provides, we generally recommend using it. In this chapter we will give recommendations on how to develop a control layer in order to benefit from its advantages.

Implementation

For the implementation of the control layer we recommend creating an own project for each controller, like e.g. a controller project for the text editor controller. This helps keeping the application flexible for future changes and raising the reusability of the components.

Service references

As for the references to the services we recommend using Spring DM for the wiring. This avoids the need to implement the service handling manually and massively increases the configurability.

Notification For the notification of the UI components in case of domain model changes we recommend implementing the Whiteboard Pattern *[Kriens04]*. A listener class should implement an appropriate and existing listener interface and should be registered in the OSGi service registry. Here, we also recommend using Spring DM for the service registration as well as for the injection of the dependencies.

Parser Although a parser is used in the UI components, we recommend placing it in the control layer because a parser generally has no dependencies to RCP. It acts more as a utility which supports the control of the user interface.

As for the concrete implementation of the parser, we cannot make any recommendation, because it depends on the requirements of your application.

Alternatives There are a lot of alternatives possible to solve this issue. We gave our recommendations based on our experiences during this thesis. The choice of an approach to solve this issue highly depends on the requirements of the application to be developed and should be specified individually.

8.5 OSGi Services

Intention This chapter contains instructions, recommendations and best practices for the definition and implementation of OSGi managed services as part of the reference architecture.

8.5.1 Distribution of service interfaces and implementations

Overview The distribution of service interfaces and implementations is a concern that a developer has to cope with when implementing OSGi services. It is an issue which has to be considered individually depending on the existing application scope. The distribution strategy for the service bundles directly affects the adaptability and the complexity of the whole OSGi service layer. Generally, there is no perfect solution for every case. The practice shows that for specific conditions there are solutions that are more appropriate than others.

Therefore, in this chapter, we will concentrate to give recommendations by also describing the circumstances in which they make sense and as well mention reasonable alternative solutions.

Recommendation Basically, the choice of how the interfaces and implementations are divided in different bundles or if they are divided at all is a trade-off between high adaptability and low maintenance effort.

Therefore the choice of the design highly depends on the application scope and the current circumstances. Because we cannot give a recommendation for a general solution we will give recommendations for different conditions in the following parts of the documentation.

For a detailed description of the possible solution alternatives and their advantages and disadvantages, we would like to refer to chapter 6.5.1.2.

Applications Having an application with a UI, most likely the control layer of it will be the di-

with a UI

rect service client which looks up and binds the services. Also, a control layer usually covers multiple functionalities provided in the user interface. Surely, this depends on the implementation model used for the UI, but nonetheless it gives a realistic context where the needs of such a service client get apparent.

For this reason, such a service client will require to import right a bunch of service interfaces for the lookup of the service instances in the service registry. In such a case, we recommend creating a bundle, which includes all the provided service interfaces for the application. Like this, the maintenance effort can be decreased and the individuality still can be held high - e.g. by grouping the service interfaces by their main concerns in multiple bundles.

The service implementations should be kept in separate bundles to assist the interchangeability. Also, we recommend having an own bundle for each service implementation in order be able to version the implementations individually.

Applications with lots of services

A large amount of services in an application can be caused by the high detail level of the service implementations and by the size of the application domain itself. The maintainability can become a very important factor in such a case.

To improve the interchangeability, we recommend placing the service interfaces and service implementations in different bundles.

Having a service interfaces bundle, which contains multiple service interfaces, helps lowering the maintenance effort. By grouping the interfaces in their main concerns, like e.g. by putting logging service interfaces in a logging bundle, this effect can be increased.

Concerning service implementations, we recommend using and own bundle for each service implementation. As already mentioned before, it helps raising the individual versioning and changeability of the concrete implementations as they are more willing to change.

Another hint we can give for such circumstances is the inclusion of default service implementations inside of the service interfaces bundle. At bundle startup, the default service implementations can be registered in the service registry. This can help decreasing the amount of bundles in your environment and thus improving the maintainability.

Applications with few services

In an application where only a small amount of services are available, maintainability does not appear in such a degree as in an application with a lot of services. Therefore the design choice highly depends on the application scope and it is really difficult to make recommendations in such a case.

Keeping in mind that in future the service count in the application could raise due to added functionality, in practice it has proven to be good to choose a scalable design which can be easy adapted to the new circumstances. This leads to the same recommendation, as for applications with a lot of bundles. It is explicitly described in the bloc above.

If the amount of bundles has to be considered, another solution implying fewer bundles can be chosen. Placing the service interfaces and the service implementations in one bundle is an alternative which could be used in such a case.

8.5.2 Implementation of OSGi services

Overview

After choosing an appropriate design for the distribution of service interfaces

and their implementations, another concern which a developer has to cope with is the implementation of OSGi services.

Implementing a service is creating an implementation class and implementing the concrete service functionality in the methods specified by the service interface. To make the service available, it has to be registered in the service registry. Also, the service client has to look up and bind the service in order get a service instance.

Considering the fact, that the implementation of such publish-find-bind procedures mostly is solved the same way, a solution allowing the elimination of duplicate code as much as possible has to be chosen.

Recommendation

For the implementation of OSGi services we recommend using Spring Dynamic Modules *[SpringDM]*. Spring DM allows the use of Spring *[Spring]* to wire the dependencies for bundles that are deployed in the OSGi environment.

Using Spring DM a developer only has to implement the service and then configure the application context in the XML description files properly. At bundle startup Spring DM takes the responsibility to create and register the service. The following code snippet describes a service publication with Spring DM, where the class `DomainServiceImpl` implements the service interface `DomainService`.

```
<!– Service implementation -->
<bean name="domainServiceImpl"
  class="org.herasaf.tutorial.personlocator
        .service.domain.DomainServiceImpl">
  <property name="listeners" ref="domainListeners" />
</bean>

<!– Service publication -->
<osgi:service id="domainService" ref="domainServiceImpl"
  interface="org.herasaf.tutorial.personlocator
             .service.DomainService" />
```

Listing 214: Code snippet showing a service publication with Spring DM

The same applies for the service client. In the configuration of the application context a service instance can be referenced through the associated service interface. At bundle startup Spring DM takes the responsibility to look up and bind the service. The following code snippet shows how a service can be referenced with Spring DM.

```
<osgi:reference id="domainService" cardinality="0..1"
  interface="org.herasaf.tutorial.personlocator
             .service.DomainService" />
```

Listing 215: Code snippet showing a service reference with Spring DM

The major advantage of this solution is that you don't have to cope with service lookup and availability issues. Also, there is no need to implement a bundle activator, which eliminates the necessity of code changes in the activator due to configuration changes.

Alternatives

The only alternative that can be mentioned here is the implementation of services using the standard OSGi libraries.

We recommend considering this alternative in cases where the Spring DM and its libraries cannot be used and where the full functionality of OSGi is required.

8.6 Utilities

Intention

In this chapter instructions and recommendations for the definition and implementation of utilities are contained.

8.6.1 Logging

Overview

The logging of application states, occurred errors and further events is an important feature of today's applications, because it allows fast error analysis as well as finding wrong behavior. Otherwise it is always required to reproduce the errors to understand it, which leads to a higher effort.

8.6.1.1 Structure

Structure

Based on the comparison of the different concepts and logging frameworks in chapter 6.6.1.1, we recommend you to use the OSGi Log Service. It offers through its structure the widest flexibility for adaptions and further extensions. Additionally there are no further bundles and plug-ins required, because the whole functionality is part of the OSGi framework.

For an easier configuration and use of the Log service, we recommend you to implement a façade bundle, like shown in **Figure 48** below. Otherwise each bundle or plug-in who wants to log something, must look up the log service. Through an indirection with the façade bundle this can be minimized. Additionally it is possible to convert and prepare the information before they are logged.

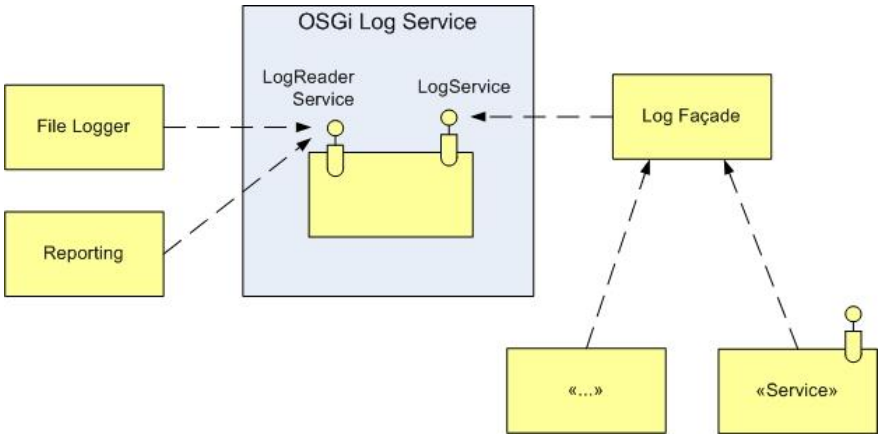

Figure 48: Schematic overview about the proposed OSGi Log service structure

8.6.1.2 Implementation

Introduction

For the implementation we recommend you to create two new bundles. One contains the log façade, which invokes the log service; the other one contains the log readers processing the log statements, for example to write them in a log file or in a database.

The reason for this recommendation is mainly to separate the log statement creation from their processing, because they have different concerns. Additionally such an encapsulation allows a flexible extension or even a replacement of one of the bundles through substitutes. Otherwise always the whole handling would be replaced.

Log façade

The implementation of the log façade class is very simple. You can create a conventional class. Now you have to implement four methods `setLogService`, `unsetLogService`, `activate` and `deactivate` with the signatures shown in **Listing 216** below. The set und unset methods are used to inject the `LogService` instance, whereas the other methods' purpose is to process activation and deactivation handling.

How you implement the remaining parts of the class is up to you. We recommend you to implement further methods which allow the creation of log statements. You can do this in this class or even better in a derived one, so you can reuse the setup and shutdown routines for several implementations.

```
public class Logger {

  private LogService logService;

  public void debug(String message) {
    logService.log(LogService.LOG_DEBUG, message);
  }

  public void debug(String message, Object arg) {
    logService.log(LogService.LOG_DEBUG,
        MessageFormat.format(message, arg));
  }

  public void debug(String message, Object[] args) {
    logService.log(LogService.LOG_DEBUG,
        MessageFormat.format(message, args));
  }

  public void setLogService(LogService logService) {
    this.logService = logService;
  }

  public void unsetLogService(LogService logService) {
    this.logService = null;
  }

  public void activate(ComponentContext context) {
  }

  public void deactivate(ComponentContext context) {
  }
  ...
  }
}
```

Listing 216: Example implementation of the `Logger` class in the log façade

Log façade configuration

Now it is required to configure them, otherwise the LogService instance is not available in the class. For this purpose you have to create a new folder *OSGI-INF* on the same level as the *META-INF* folder. In this folder you have to create a new file *component-description.xml*, which contains the component description. It creates an instance of the log façade class and injects an instance of the `LogService`.

The code snippet in **Listing 217** below shows such a component description.

```
<?xml version="1.0"?>
<component name="logger">
  <implementation
      class="org.herasaf.tutorial.personlocator.log.Logger" />
  <reference name="logService"
      interface="org.osgi.service.log.LogService"
      bind="setLogService" unbind="unsetLogService" />
</component>
```

Listing 217: Component description of the Log service

Additionally you have to declare the component description in the bundle manifest with the header `Service-Component`. Otherwise the OSGi framework doesn't know that an instance of the `LogService` is required and cannot inject them.

```
Service-Component: OSGI-INF/component-description.xml
```

Listing 218: Additional header in the bundle manifest for the component declaration

Log reader

Very similar but not equals is the implementation of the log reader. We recommend you to implement it using a `LogListener`, because you will be notified about new log statements. Otherwise you have to poll regularly if new statements are available. But this would lead to implementations with own threads and synchronizations, which should be avoided as much as possible.

Also here you have to implement four methods. The first two are used for setting and unsetting the `LogReaderService` instance. The other ones are for activating and deactivating the component. You can use them to register and deregister the listener from the `LogReaderService`.

The last method which is required is `logged`. It is invoked from the `LogReaderService` in order to notify about new log statements. In this method you can implement the specific processing of the log statements.

Listing 219 shows such an implementation using a `LogListener` to be notified about new log statements.

```
public class ConsoleLogListener implements LogListener {

  protected LogReaderService logReaderService;

  public void setLogReaderService(LogReaderService logReader) {
    this.logReaderService = logReader;
  }

  public void unsetLogReaderService(LogReaderService logReader) {
    this.logReaderService = null;
```

```
  }

  protected void activate(ComponentContext context) {
    logReaderService.addLogListener(this);
  }

  protected void deactivate(ComponentContext context) {
    logReaderService.removeLogListener(this);
  }

  public void logged(LogEntry entry) {
    // Specific handling of the LogEntry
  }
}
```

Listing 219: Example implementation of a `LogListener` using the `LogReaderService`

Log reader configuration

Also for the same reasons like for the Log service, you have to define a component description in the folder *OSGI-INF*, which creates an instance of the log reader class and injects an instance of the `LogReaderService`.

The code snippet in **Listing 220** below shows such a component description.

```
<?xml version="1.0"?>
<component name="consoleLogListener">
  <implementation
      class="org.herasaf.tutorial.personlocator.logtracker.
             ConsoleLogListener" />
  <reference name="logService"
      interface="org.osgi.service.log.LogReaderService"
      bind="setLogReaderService"
      unbind="unsetLogReaderService" />
</component>
```

Listing 220: Component description of the LogReader service

Also this component description must be declared in the bundle manifest with the header `Service-Component`. Otherwise the OSGi framework doesn't know that an instance of the `LogReaderService` is required and cannot inject them.

Configuration with Spring DM

As an alternative to the component descriptions it is also possible to get the `LogService` and `LogReaderService` instances through Spring DM. We recommend you this solution especially in cases where you wire the remaining parts of the application with Spring. In other cases you have no further advantage.

8.7 Test

Intention

This chapter contains instructions, recommendations and best practices for the development of unit and integration test used for the verification of the components in the reference architecture.

8.7.1 Unit Test

Overview

As already mentioned before, in the description of our solution, the unit tests are in a scope outside of the OSGi environment, assuring functionality inside of the bundle's bounds. Therefore, to create and run the tests common testing frameworks can be used.

This part of the documentation will give recommendations and hints how to create, integrate and run unit tests for your bundles.

Recommendation

We recommend implementing your unit tests in **separate projects**. Having the unit tests in the same project as the bundle's project has shown problems with duplicate entries in the *Maven Classpath Container* and the *Plug-in Dependencies Container* when referencing other projects. Another problem is the visibility of Maven artifacts inside of OSGi bundles. Referring to these problems, no proper separation of the containers inside one project could be realized.

As for a test framework, we recommend using **TestNG** *[TestNG]*. TestNG is an advanced testing framework introducing new functionalities to make it more powerful and easier to use than common JUnit. Amongst others it supports Spring, data-driven testing with *DataProviders*, easy configuration through Annotations and is supported by various tools and plug-ins like Eclipse, Maven, etc. Having Spring handling your beans in combination with TestNG makes defining tests very simple and transparent. It has proven to be very efficient in bigger projects when you have a lot of unit tests and test data to be covered.

We also recommend **integrating** the unit tests in the **building environment**, like e.g. Maven. This can be done easily considering the fact that it's a question of conventional unit tests to be compiled and executed. Using Maven and integrating your unit tests in the automatic build has proven to give transparency and stability to your product.

8.7.1.1 Implementation

Intention

Referring to the recommendation above, in this part we show you how to implement a unit test using TestNG and Spring framework.

A detailed tutorial on how to create a unit test step-by-step is available in chapter 16.4. It's recommended to read the tutorial before you continue reading this part.

Implementation

The TestNG test framework supports the creation of tests based on a Spring application context. To implement such a test an extension of the class `AbstractTestNGSpringContextTests` has to be defined. Using the *@ContextConfiguration* annotation, the application context descriptor can be specified. The following code snippet gives an example of a test.

```
@ContextConfiguration(
    locations={"classpath:/applicationContext.xml" })
public class TranslationServiceImplTest extends
    AbstractTestNGSpringContextTests {

  @Autowired
  private TranslationServiceImpl service;
```

```
  @Test
  public void testAvailableText() throws Exception {

    assertNotNull(service);

    String key = "appName";
    String translation = "OSGi Tutorial Service";
    assertEquals(translation, service.getTranslation(key));
  }
}
```

Listing 221: Code Example of a TestNG unit test using a Spring application context

By using the *@Autowired* annotation, required fields in the test class can be defined. The Spring framework will look for an applicable object to be injected during the creation of the application context. Please note that the developer has to ensure the availability of an appropriate object with the correct type in the application context. Otherwise the Spring framework won't be able to find an appropriate object to inject into the annotated attribute.

As for the test methods, the *@Test* annotation can be used to specify them. Using annotations allows free choice of test method names, since the identification of them isn't based on the method name itself.

Configuration

Considering the use of TestNG *[TestNG]* and Spring *[Spring]* framework, we'd like to refer to the websites of the projects and their documentation for a detailed description of the possibilities related to the configuration and setup.

8.7.2 Integration Test

Overview

Integration tests are integral components for the application verification during the whole development and maintenance phases. They allow every time not only the tests of single bundles but rather to verify the dependencies between them.

A correct and uniform implementation of them allows additionally running them in a continuous integration environment. That's why we want to take a deeper look at them in this chapter.

Recommendation

To achieve the uniformity of your integration tests, we give you in the following part some recommendations and hints, which proved to be very useful.

We recommend you to implement your integration tests in **separate projects** beside the several bundles and components of your application. There are two reasons for this procedure. The first one is the missing affiliation to exactly one of your bundles or components. So it's inappropriate to implement them in one of your other projects.

The second reason is accompanied with further recommendation to you. With the use of the **Spring DM testing framework**, described in detail in the Spring DM reference guide *[SpringDM]* the integration test implementation is different to them of normal bundles or components. The reasons for this are on the one hand the simple development, because you can implement normal Java projects, on the other hand the support of an integrated scenario by the Spring DM testing framework like the extract of the reference guide *[SpringDM]* below

shows.

- *Starts the OSGi framework.*
- *Install and start any specified bundles required for the test.*
- *Package the test case itself into an on-the-fly bundle, generate the manifest (if none is provided) and install it in the OSGi framework.*
- *Execute the test case inside the OSGi framework.*
- *Shut down the framework.*
- *Passes the test results back to the originating test case instance that is running outside of OSGi.*

If you decide to implement your integration tests without the Spring DM testing framework, you have to implement or automate these steps on your own. That's why we recommend you to use the testing framework. Through its simplicity you can concentrate on the real problems of your application.

You can find a detailed comparison of several integration test strategies, including the Spring DM testing framework, in chapter 6.7.2. You may take a look at them, before you decide.

Additionally we recommend you to **embed** your integration tests **in your build environment** like Maven. It allows you in later correction or extension steps the verification of existing parts without any extra effort. There by it increases the stability of your system and increases the transparence against your principal.

8.7.2.1 Implementation

Intention

After our recommendation we show you now, how you can implement an integration test using the Spring DM testing framework and the Maven build automation.

A tutorial implementing such an integration test is available in chapter 16.5. Take a look at it before you continue to reading this chapter.

Implementation

The implementation of an integration test is very similar to a JUnit test. You have to extend the existing class `AbstractConfigurableBundleCreatorTests` and implement your own test methods with the following signature.

```
public void testxxx() throws Exception;
```

Now you can implement your integration tests in the well-known manner using the assert methods provided by the abstract class.

8.7.2.2 Configuration & Customization

Intention

This is only one side of the medal; the other is the configuration and customization, which can be done only programmatically.

The base classes of the Spring DM testing framework offer you additionally a wide range of configuration and customization possibilities. They can be used by overwriting defined methods. The use of annotations for that purpose is at the moment this thesis was written not available, but is intended in future

versions.

Now we take a look at some configurations and customizations for your integration tests. We show only a selection of main configuration and customizations. For further information, see the Spring DM reference guide *[SpringDM]* or corresponding literature.

Register bundles

One of the main configurations is the registration and usage of the bundles defining the target platform.

You can get access to bundles in your integration tests through the variable `bundleContext`. You can use it in the same manner like the `BundleContext` variables in the activator classes of your bundles. You can register bundles and get references to them.

Although the Spring DM testing framework locates the bundles for the target platform by default out of the local Maven repository you have to define which ones should be installed and started. By default only a handful are automatically detected. One of them is the bundle *org.eclipse.osgi* from Equinox or the corresponding bundle of another platform.

To allow the location of further bundles, you must override the method `getTestBundlesNames`. The method returns a String array with a Maven-like description consisting of *GroupID*, *ArtifactID* and *Version* for the required bundles. An example is shown below.

```
@Override
protected String[] getTestBundlesNames() {
  return new String[] {
    "org.herasaf.tutorial.service, tutorial-osgi-service,
1.0.0",
    "org.herasaf.tutorial.service.impl, tutorial-osgi-service-
impl, 1.0.0" };
}
```

Listing 222: Example of the `getTestBundlesNames` method out of an integration test

Like mentioned above by default the local Maven repository is used. At the moment this thesis was written only this implementation existed, but further ones are intended.

If you want to use an own locator to locate bundles in other repositories, you can develop an additional one by implementing the interface `org.springframework.osgi.test.provisioning.ArtifactLocator`. For further details, see the Spring reference guide *[SpringDM]* or corresponding literature.

Manifest manipulation

Another important configuration respectively customization is the manifest manipulation. The testing framework offers you either the manipulation of a generated or the creation of an own manifest.

The manipulation can be achieved by overwriting the method `getManifest`. In a first step, you have to generate the base manifest through the invocation of the same method in the super class. Afterwards you can manipulate its values like shown in the example below.

```
@Override
```

```
protected Manifest getManifest() {

  Manifest mf = super.getManifest();
  mf.getMainAttributes().putValue(Constants.BUNDLE_CLASSPATH,
".,lib/log4j.jar");

  return mf;
}
```

Listing 223: Example of the `getManifest` method out of an integration test

Another possibility to manipulate is to define one on your own and to instruct the testing framework to use it. You can achieve this by overwriting the method `getManifestLocation`. Its return value is the path to the manifest file so that it can be used instead of a generated one. You can use the *classpath* syntax which should be known from the Spring Framework. The example below shows such an implementation.

```
@Override
protected String getManifestLocation() {
  return "classpath:META-INF/MANIFEST.MF";
}
```

Listing 224: Example of the `getManifestLocation` method out of an integration test

Use of an ApplicationContext

The Spring DM testing framework allows also the use of the typical Spring configurations for dependency injection and further more. Instead the use of annotations to link it with the test class you have to overwrite the method `getConfigLocations`. Its return value is a String array returning the paths of the configuration files used for the integration test. An example is shown below.

```
@Override
protected String[] getConfigLocations() {
  return new String[] { "/applicationContext.xml" };
}
```

Listing 225: Example of the `getConfigLocations` method out of an integration test

The application context is then accessible in your test class through a variable called `applicationContext`.

Platform configuration

After the configuration of the test cases, you can also configure the platform and the lifecycle of the integration testing.

Through the overwriting of the method `getPlatformName` you can declare which platform should be used for testing. The return value is a String with the name of the platform. You can use the constants in the class `Platforms` for this purpose. By default the value of the system property `OSGI_FRAMEWORK_SELECTOR` will be used. An example is shown below.

```
@Override
protected String getPlatformName() {
  return Platforms.EQUINOX;
}
```

Listing 226: Example of the `getPlatformName` method out of an integration test

The other configuration concerns the lifecycle of the integration test. With the method `shouldWaitForSpringBundlesContextCreation` you can force the testing framework to wait for the context creation of Spring/OSGi bundles before executing the tests. Its return value is a Boolean and *true* will be returned by default.

For further configurations and adaptations consult the Spring DM reference guide *[SpringDM]* or corresponding literature.

8.7.2.3 Hints

Several test classes

The development of integration tests, implemented in several test classes, may cause unexpected exceptions during the execution. The causes of this problem are the test configurations, covering the manifest, the application context and the dependent bundles. It seems the Spring DM testing framework is not able to handle the execution of more than one test class if they have different configurations.

To solve this problem we recommend you to implement an abstract super class defining the manifest, application context and dependent bundles for all test cases. Afterwards you can inherit your specific test classes from the abstract one. Such an implementation ensures that all test cases have the same configurations. Otherwise you have to implement several projects containing only one integration test class.

Obviously this solution has the disadvantage that the configurations cannot be separated for the different test classes. So you are required to implement the test classes in a properly manner and to keep the configuration files clean.

8.8 Configuration

Intention

In this chapter instructions and recommendations for a correct and flexible configuration of the components in the reference architecture are contained.

8.8.1 OSGi Manifest

Bundle Description

According to the OSGi Core Specification *[OSGiCore]* a bundle is a JAR file containing the necessary resources to provide some functionality. It also contains a manifest file MANIFEST.MF describing the whole bundle. The manifest file is specified to be located in the META-INF directory. Optional content is located in the OSGI-OPT directory and can be e.g. documentation and/or the source code.

Manifest Headers

The content in the manifest file carries descriptive information about the bundle itself in form of key-value pairs. The specification determines a large list of possible headers, matching attributes and restrictions. A detailed description of all specified manifest headers can be found in the OSGi Core Specification *[OSGi-*

Core] under the chapter “3.2.1 Bundle Manifest Headers”.

In practice only a subset of the specified manifest headers is needed to fulfill the requirements for a proper description of the bundles. The following list describes the recommended manifest headers to be used (mandatory fields are bold).

Manifest Header	**Description**
Bundle-ActivationPolicy	Specifies the activation policy for the bundle. There is only the *lazy* activation policy defined, meaning that if no *Bundle-ActivationPolicy* header is specified, the bundle will use eager activation. Example: Bundle-ActivationPolicy: lazy This header is optional. A lazy activation policy means that the bundle will be started but not activated until a class is requested from it. By using lazy activation the initialization time during the startup can be shortened and resources can be potentially saved. However to be sure how this affects the behavior an analysis is essential.
Bundle-Activator	Specifies the Bundle Activator for the bundle. An activator class must implement the *BundleActivator* interface, be declared public and have a default constructor since reflection is used to create a new instance with *Class.newInstance*. Example: Bundle-Activator: ch.hsr.calculator.core.Activator The *Bundle-Activator* header is optional. If no *Bundle-Activator* header is specified, no Activator will be executed even if there is a class in your bundle implementing the *BundleActivator* interface. In some cases, where behavior is needed during the startup and shutdown of a bundle, an activator gives the opportunity to react on these points of time. Having a small bundle though only exporting packages no activator definition is needed.
Bundle-Classpath	Defines a list of comma separated file path names or directories of JAR files inside the bundle, which contain classes and resources. The period character ‘.’ represents the bundle’s JAR file itself. Example: Bundle-Classpath: /lib/lib1.jar,/lib/lib2.jar,… This header is optional. The period is the default setting if no *Bundle-Classpath* header is specified. The use of this header is only recommended if there are specific libraries needed to run the bundle.
Bundle-Copyright	Contains the copyright for the bundle. Example:

	Bundle-Copyright: HERAS^{AF} (c) 2009 This header is optional. It's recommended to specify the *Bundle-Copyright* if one is used and available in order to know where the bundle belongs to.
Bundle-Description	Defines a description for the bundle. Example: Bundle-Description: Calculator Core Bundle This header is optional. It's recommended to define a short description to know what the bundle is about without the need of further analysis of the bundle.
Bundle-DocURL	Contains an URL referencing to the documentation of the bundle. Example: http://www.herasaf.org/ercpra/doc This header is optional. If a documentation for the bundle is available it's recommended to specify it here.
Bundle-ManifestVersion	Declares that the bundle follows the rules of the [OSGiCore] Specification. Example: Bundle-ManifestVersion: 2 For Release 4 bundles and later, number 2 is specified, whereas for Release 3 bundles, number 1 is declared. Even though this header is optional, it's highly recommended to define the *Bundle-ManifestVersion* in order to keep clarity which Specification is considered.
Bundle-Name	Specifies the name of the bundle. Example: Bundle-Name: Calculator Core This header is mandatory. It gives the bundle a readable name and identification.
Bundle-SymbolicName	Specifies a unique name for the bundle. The bundle symbolic name and the bundle version allow a bundle to be identified uniquely in the framework. Example: Bundle-SymbolicName: ch.hsr.calculator.core The *Bundle-SymbolicName* header is mandatory.
Bundle-RequiredExecutionEnvironment	Specifies a comma separated list of execution environments that must be present in the Service Platform. Example: Bundle-RequiredExecutionEnvironment: J2SE-1.5 Even though this header isn't mandatory, it's strongly recommended to define it. The definition of

	required execution environments allows a bundle to require a specific environment on a framework before being installed and gives information about the execution environments a framework provides.
Bundle-Vendor	Specifies the description of the bundle vendor. Example: Bundle-Vendor: HERAS[AF] This header is optional. We recommend using this header if a vendor is available and possible to specify.
Bundle-Version	Specifies the version of the bundle. Example: Bundle-Version: 1.0.0 The *Bundle-Version* header is mandatory.
DynamicImport-Package	Specifies a comma separated list of package names to be dynamically imported when needed. Example: DynamicImport-Package: ch.hsr.calculator.* This header is optional. It's recommended using dynamic imports as last resort because they're matched to export definitions during class loading and do not affect the module resolution. More details on this can be found in the *[OSGiCore]* Specification under "3.8.2 Dynamic Import Package".
Export-Package	Specifies a comma separated list of exported packages. Example: Export-Package: ch.hsr.calculator.core.model This header is optional. It should be applied on all packages intended to be available and visible from outside.
Fragment-Host	Specifies the host bundle for "this" fragment. Example: Fragment-Host: ch.hsr.calculator.core This header is optional. The declaration of this header only makes sense when working with fragment bundles. In that case it's recommended to define the host bundle for the fragment so that the framework knows where to attach it. A detailed description of this process can be found in the *[OSGiCore]* Specification under "3.14.1 Fragment-Host".
Import-Package	Specifies the imported packages for this bundle. Example: Import-Package: org.osgi.service.io;version=1.4 This header is optional. Generally it's preferred to use the *Import-Package* and *Export-Package* headers to wire bundles. The coupling between the

	importer and exporter is lower. The package composition of bundles can be refactored without causing other bundles to fail.
Require-Bundle	Specifies the required exports from another bundle. Example: Require-Bundle: org.eclipse.ui This header is optional. The *Require-Bundle* header binds all the exports of another bundle, regardless of what the exports are. This can lead to *Split Packages* and/or other problems. Detailed information about this can be found in the *[OSGiCore]* Specification under “3.13.3 Issues With Requiring Bundles”.

Table 4: Description of the specified manifest headers

Please note that the *[OSGiCore]* Specification specifies that unrecognized manifest headers are ignored by the Framework implementation. This means, that the bundle developer can define custom additional manifest headers if required.

8.8.2 Bundle dependency types

Overview

During the development of applications it’s obvious to use classes and interfaces out of other libraries. In an OSGi environment a bundle can use not only libraries but also other bundles and plug-ins. To use the classes out of them, they must be imported in such a way through a directive in the bundle manifest file. Otherwise the classes and interfaces are not accessible.

Therefore the OSGi compliant manifest provides different kinds of import directives. The use of them and especially the decision which one has direct impact on the adaptability and stability of your bundle or plug-in.

Recommendation

Based on our own experiences described in chapter 6.8.1 and the suggestions from different authors like in the OSGi book *[Wütherich2008]*, we recommend you to use primarily the *Import-Package* and *DynamicImport-Package* directives. The reason for this recommendation is mainly the flexibility you gain, because you don’t have a direct dependency to a specific bundle, only to the packages they export. Therefore the bundles providing these packages can exchanged easily without a change in your own bundle.

Alternatives

But in special cases, for example when you have problems with split packages or you are using a special bundle anyway, you can import the packages also with the *Require-Bundle* directive. Compared with the simple package import directives, you import here all classes and interfaces, not only the exported ones. This can be an advantage if you need access to the private classes or extension points of bundle.

So we recommend also the use of the *Require-Bundle* directive but appeal for a careful usage. Consider the advantages and disadvantages in the concrete situations to get the appropriate solution for them.

8.8.3 Maven

Overview

In this chapter Maven related configurations are described.

8.8.3.1 Maven Bundle Plug-in Configuration

General

The Maven Bundle Plug-in configuration is defined in the *configuration* part of the plug-in definition contained in the pom.xml. There you can define plug-in specific *instructions* giving it the needed parameters to do how it's best suited for you.

It's recommended to define *properties* in your pom.xml to ensure writing the entries only once and in one place. If you have more projects you can define general *properties* in the main projects pom.xml. This acts as a parent for the other modules and the defined *properties* can be used in all sub projects.

Plug-in instructions

The code snippet below shows the defined *instructions* to configure the Maven Bundle Plug-in. They are inside the *instructions* tag, which is embedded in the *configuration* tag of the plug-in. As tag values they use references to predefined *properties*. The tags itself correlate with the entries in the MANIFEST.MF file of the OSGi bundle, see chapter 8.8.1.

```
<instructions>
  <Bundle-Name>${artifactId}</Bundle-Name>
  <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
  <Import-Package>${service.import}</Import-Package>
  <Export-Package>${service.export}</Export-Package>
  <Private-Package>${service.private}</Private-Package>
  <Bundle-Activator>${service.activator}</Bundle-Activator>
  <Require-Bundle>${service.require}</Require-Bundle>
  <Include-Resource>${service.resource}</Include-Resource>
  <Bundle-RequiredExecutionEnvironment>
    ${service.environment}
  </Bundle-RequiredExecutionEnvironment>
</instructions>
```

Listing 227: Snippet of the instructions tag in the maven-bundle-plugin configuration

Please note that this is only a subset of possible *instructions*. There are many more tags which can be used to configure this plug-in. More information can be found on the webpage of the project *[ApacheFelix]*.

Properties

The code snippet below shows the various *properties*. The tag names inside the *properties* tag are self-defined and get used as the key when referencing to them. They simply get resolved and the value inside the tags gets copied to that place where they are referenced.

```
<properties>

  <manifest.private>
    org.herasaf.ercpra.activator,
    org.herasaf.ercpra*impl*
  </manifest.private>
```

```
  <manifest.export>
    org.herasaf.ercpra.service
  </manifest.export>

  <manifest.import>
    !${service.export},
    *;resolution:=optional,
    org.osgi.framework;version="1.3.0"
  </manifest.import>

  <manifest.activator>
    org.herasaf.ercpra.activator.ServiceActivator
  </manifest.activator>

  <manifest.require>
    org.springframework.bundle.osgi.core;bundle-version="1.1.3",
    org.springframework.bundle.osgi.extender;bundle-
version="1.1.3",
    org.springframework.bundle.osgi.io;bundle-version="1.1.3"
  </manifest.require>

  <manifest.resource>
    META-INF/spring=META-INF/spring
  </manifest.resource>

  <manifest.environment>J2SE-1.5</manifest.environment>

</properties>
```

Listing 228: Snippet of the properties used to configure the maven-bundle-plugin

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Part IV: Deployment

9 Overview

Introduction

An automatic build environment is a very obvious part for the compilation and creation of a simple Java archive as well as a complete product. In this chapter the dependencies to libraries, the automated execution of tests and the whole deployment of the reference architecture is described.

Maven2 *[Maven]* is described detailed in this part because it acts as the base tool for the automatic build. The information about how to configure Maven2 and its plugins is spread over multiple web pages and books. Like this, the information is collected at one point.

For the continuous execution of the automatic build and its test cases on the server side, Atlassian Bamboo *[Bamboo]* is used. Bamboo is an integration server automating the process of compiling and testing source code. It provides reports and statistics to help saving time and to recognize failures and problems in an easy way. Bamboo is widely used in the HERAS^{AF} project. Therefore it was already installed and set up properly at the start of this bachelor thesis.

In a first part an overview of Maven2 and the used plugins is given followed by an explanation of its extension with further plug-ins.

10 Apache Maven 2

Introduction

The proper installation and configuration of Apache Maven 2 is required in order to be able to use it as an automated software build solution.

settings.xml

The *settings.xml* contains the configuration settings for Maven. An example of such a setting is where the local-repository is located or which profiles are defined, and so on.

POM

The *pom.xml* files represent the Project Object Model and are fundamental of working with Maven. Written in XML it holds information about the project and the configuration details used to build the project.

External libraries

External libraries can be loaded by declaring it as a dependency in the *pom.xml* description file. If the declaration is correct, Maven will load the appropriate library in the requested version.

Project dependencies

The *pom.xml* description file also includes the project dependencies. The declaration of them takes place analogue to the external library declaration in a dependency tag.

Tests

Maven declares the folder `/src/test/java` as its default source directory for test classes. Every project can contain tests located in this folder.

10.1 Plugins

Introduction

Maven is designed as an execution framework of plugins. All the work is done by plugins. This chapter lists the plugins used for the development of the HERAS^{AF} PIP.

Javadoc generation

The maven-javadoc-plugin *[MVNJavaDoc]* is provided to generate javadoc and pack it as a jar.

```
<plugin>
  <artifactId>maven-javadoc-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <aggregate>false</aggregate>
    <quiet>true</quiet>
  </configuration>
</plugin>
```

Listing 229: Example maven-javadoc-plugin plugin configuration

Important here is to define the execution phase tag as `package`. Using other phases can cause *NullPointerException*s during the javadoc generation.

Source packing

The maven-source-plugin *[MVNSource]* is provided to create a jar archive of the source files.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-source-plugin</artifactId>
  <executions>
    <execution>
      <id>attach-sources</id>
      <phase>verify</phase>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 230: maven-source-plugin plugin configuration

Source compiling

The maven-compiler-plugin *[MVNCompiler]* is provided to compile the sources in the projects.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
```

```
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

Listing 231: maven-compiler-plugin plugin configuration

According to an error free compilation and build it's important to declare the correct version of the compiler. The default compiler used by this plugin is the `javac` compiler.

Bundle Plug-in for Maven

The maven-bundle-plugin *[ApacheFelix]* is provided to handle the manifest generation for OSGi bundles as well as building and delivering bundles out of the project.

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.0.0</version>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      ...
    </instructions>
  </configuration>
</plugin>
```

Listing 232: maven-bundle-plugin plugin configuration

For a correct generation of the manifest file, the instructions have to be configured in sufficient scope; otherwise the bundle cannot be activated in the OSGi runtime environment.

11 Hints

Introduction

During this bachelor thesis some experiences were made working with Maven and its plug-ins. This chapter is thought to give some hints and to mention the discoveries made.

11.1 Maven and Eclipse

Q4E Plugin

The Q4E Plugin *[Q4E]* is provided by Google and allows using Maven 2 from the Eclipse IDE.

There are some useful features provided with this plugin like e.g. the Dependency Analysis which helps to clean up the dependencies of the modules. Using this plugin library version conflicts occurring during the implementation could be fixed.

A major disadvantage of this product is that it is still in its beginning phase and contains a lot of defects, which sometimes makes the work quite painful.

IAM Plugin

But some of them were fixed through the handover of the plug-in development and maintenance to Eclipse under the new name IAM *[EclipseIAM]* (Integration for Apache Maven).

Maven build and errors

Sometimes it happens that Eclipse suddenly shows errors in the projects. This is even caused by a Maven build or by the updating classpath process Maven executes each time Eclipse is started. A solution for this problem is performing a `clean` on the relevant projects. If this doesn't help, try reverting project files changed during the update classpath process from the SVN repository.

In the case of strange errors appearing and none of the hints above are successful; try deleting your local projects from Eclipse and re-checkout them from the SVN server. Your changes should be checked in before carrying out this step.

Errors after Maven build

It also possible that Eclipse shows errors after a successful Maven build. The cause of these problems is not the Maven build itself but Eclipse. It seems that Eclipse is not able to correctly parse automatic generated manifest files, although the content is correct.

This can be solved through the formatting of the manifest file with the legacy formatter of Eclipse.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Appendix A: General

12 Final Report

Intention

This part of the documentation describes the project after its completion. It contains a summary of the realized parts and a report of the achieved goals. At the end, an outlook for future tasks and extensions is given.

12.1 Summary

General

Infrastructure

We started the bachelor thesis with the setup of the infrastructure. For this purpose we had to analyze the requirements of the development environment. We installed and setup Eclipse with various plug-ins, the target platform and Maven for the build process. Further, we configured Achievo, Bamboo and JIRA for the organization of the project.

Research

As a next step, we started the research on the basics of OSGi, Eclipse RCP and Spring Dynamic Modules. It includes the extensive internet research, reading books and the implementation of simple example applications. Beside the analysis of the interaction of the specific technologies, we also defined first requirements for the reference architecture.

Automatic Build

In this part we developed an automatic build environment to support and automate the development process. In a first step, we analyzed the requirements for such a solution and what tools are currently available. Based on them, we started the design of it. Important topics were how they can be configured to work together and if they are still maintained and enhanced.

After a detailed analysis and the decision for the appropriate tools, we started the iterative implementation of the automatic build process. The result is the full support of the development steps, from the generation of the manifest to the creation of deliverable products.

Beside the analysis, conception and development of the build process, we documented the solution including the design decisions and a detailed developer's guide.

Text Editor Development

After the integration of the automatic build environment, we started the development of a text editor component. We first made a research on the available Eclipse technologies in this domain. The requirements for the text editor were acquired through a detailed analysis of the existing components.

The implementation of the text editor was started after the decision which components to use. The result is a text editor supporting the user in its tasks with various features, such as syntax highlighting, outline page, and further more. Additionally, Spring and Spring DM were used for the application wiring and the component was fully embedded in the automatic build process.

During the analysis, conception and the development of the text editor, we also documented the solution and the design decisions as well as detailed instructions in the developer's guide.

Graphical Editor Development

Simultaneously, we started the development of a graphical editor. In a first step we had to analyze which technologies could be used. The Eclipse project provides two frameworks for this purpose, the Graphical Editing Framework as well as the Graphical Modeling Framework. A detailed analysis of their individual advantages and disadvantages resulted in the decision to use GEF. In a further step, the requirements for a graphical editor component were specified.

The implementation started with the definition which components will be required. Based on the gathered design, the graphical editor was realized. The result is an extensible and flexible graphical component supporting the user with various features, like palette, drag & drop and many more. Further, Spring and Spring DM were used for the application wiring and the component was fully embedded in the automatic build process.

During the development process, the documentation was updated with the description of the solution as well as detailed instructions in the developer's guide.

Integration and Refactoring

In this part, we put together the text editor and the graphical editor components into a whole rich client application with a common domain model encapsulated by an OSGi service layer. The definition of joint interfaces was already specified during the implementation of the specific components. Additionally, refactorings were made in order to fulfill the low dependency principle, to mention only one.

Further, the gained experiences during the integration and the refactoring were included in the documentation.

Finalization

In the last two weeks we finalized the documentation and prepared all the artifacts for the completion of the bachelor thesis.

Coverage

Automatic Build

The result of this part is an integrated build environment based on Maven. Through the use of Apache Felix's Bundle Plug-in for Maven OSGi specific configurations and implementations are supported. This covers beside the manifest generation also the creation of valid OSGi Bundle Archives.

Complementary, the execution of unit and integration tests inside of the OSGi environment is realized with the use of the Spring DM Testing Framework. It encapsulates the tests with pre- and post-processing. These cover amongst other steps, the generation of an on-the-fly test bundle, the setup and startup of an appropriate OSGi environment and the cleanup and teardown at the end.

The whole build process is completed with the automatic generation of deliverable RCP products. This has been achieved through the integration of Eclipse's PDE Build.

Reference Architecture

Concerning the development of a reference architecture, the following parts have been covered:

- Modular application architecture based on SOA principles
- Integration of RCP components, such as a Text Editor, Graphical Editor, Problems View and furthers.
- Definition of OSGi Services

- Synchronization of the different components in a flexible and encapsulated structure
- Wiring of the whole application using Spring, Spring DM and Spring Utilities
- Design of unit and integration tests for the verification of the application

Documentation

The documentation is the third part of the bachelor thesis. It consists on the one hand of a detailed analysis and the description of the solution. On the other hand it provides the definition of development processes and instructions for developers how to implement custom applications based on this reference architecture. The documentation includes additionally best practices and recommendations for this purpose.

12.2 Outlook

General

In this chapter further extensions and changes of the developed reference architecture are described in an aggregated form.

Technologies

Consideration of coming technologies

In this bachelor thesis, the newest available tools and libraries were used. In future implementations, though, coming technologies should be considered. Developers especially should keep an eye on the further development of the tools used in the automatic build solution as there is still room for improvement in terms of usability, simplicity and the integration of Eclipse technologies.

Continuous upgrade of used frameworks and tools

The used frameworks and tools in the developed reference architecture should be updated continuously. Through this, the developer can benefit from further implementations and thus from simplifications and more stable tools. Beside the Eclipse framework and Spring, also the tools used in the automatic build solution should be upgraded regularly.

Automatic Build

Monitoring the progress of the Tycho Plug-in

The Tycho plug-in integrates the common steps for the creation of OSGi bundles, like e.g. generation of the manifest, and a complete creation of a product using Eclipse technologies. However, the Tycho project is currently still in the development phase. In the future, the progress of it should be monitored.

Improvement of the PDE Build

In the automatic build solution, especially the PDE build leaves room for improvement. In the current solution, the Maven AntRun plug-in and Ant scripts are used to accomplish this task. In the future, though, coming technologies and new versions of used libraries and tools should be considered in order to develop a better integration of the PDE build in the automatic build process.

Reference Architecture

Integration of extended services

In the future, the integration of extended services should be considered. An

extended service can be e.g. a web-service and/or a service providing advanced access to a relational database. Such an extension broadens the horizon of the applications implemented based on this reference architecture.

Implementation and integration of further RCP components

The developed reference architecture in this bachelor thesis gives the base for the implementation of sophisticated rich client applications. In the future, the implementation and integration of further RCP components should be considered.

Switch to TestNG for the integration tests

In the current version, Spring DM only supports JUnit as a testing framework for the development of integration tests. According to Spring, an integration of the TestNG testing framework is planned in the future. Therefore, the further development of the Spring DM project should be monitored. A switch to TestNG enables the creation of advanced tests through the use of modern technologies.

Documentation

Continuous update of the documentation

The documentation is an important part of this bachelor thesis. As already mentioned before, the used technologies will be changing in the future and providing more simplifications and further functionality. Therefore and in order to simplify the use of the developed reference architecture, the documentation should be updated continuously, as well.

13 Personal Reports

13.1 Tobias Forster

Motivation

During the finalization phase of our term thesis Ylli Sylejmani and I talked with René Eggenschwiler about the possibility to do our bachelor thesis in the HERAS^{AF} project. He offered us about three topics, they were considering. The most interesting one seemed in our opinion the development of a reference architecture using OSGi and Eclipse RCP. Although we had no experiences with these technologies, we agreed on the spot, because we knew it would be a unique chance and a huge challenge. An additional stimulus was the intent, that the result of this thesis shall be used by Florian Huonder for his master thesis.

Research

So we started this thesis in mid February with a comprehensive research about OSGi, Spring DM and Eclipse RCP. In books and on websites we learned a lot about these technologies. To get familiar with them, we started the implementation of first tutorial applications.

Automatic Build

After two weeks of research and tutorials, we decided with René to support the whole development process with an automatic build. This sounded much easier than it was, because a conventional Maven build is not sufficient. It must consider the OSGi characteristics, not only for the creation of compliant bundles, but also for the execution of unit and integration tests.

Through the combination of several technologies, like the Spring DM testing framework or Maven plug-ins, we could develop a stable build process. It covered beside the creation of valid bundles also the generation of deliverable and executable products using the Eclipse PDE Build.

The presentation of this intermediate result to our expert and tutor was very impressive through the combination of the technologies, although it was a simple build process.

Graphical editor

With it, we could close an important milestone and concentrate on the next part of the reference architecture. This covered the implementation and integration of a text editor and a graphical editor. Ylli and I decided each of us should concentrate on one of the two parts. So I took the graphical editor and started with the research.

As a first step I had to analyze GEF and GMF and to decide together with Ylli which framework would be the appropriate one to satisfy our requirements. We decided for GEF with the consequence of an increased implementation effort. So I had to design the editor structure and to implement it afterwards.

Integration

During this time, we had to consider continuously the joint points for their integration, because the two components had to work together. Special challenges were in this part the management of a common domain model in an OSGi managed service and the synchronization of the components.

Documentation

While doing all these research, analysis, design and implementation tasks, we wrote and updated the documentation, which is the main result of this thesis. It contains beside the description of the solution also development processes and a developer's guide based on best practices and gained experiences. This was

a huge effort and a challenge at the same time, because it had to be clear and understandable for the reader.

Organization

This was only possible through a flexible organization of our project using an agile approach very similar to Scrum. Here especially the tools JIRA, Mylyn and Achievo gave us an optimal support. Together with the guidance through René, it allowed us to plan and organize the project optimally.

Experiences

So I could learn a lot apart from the technologies. Personally, I gained further experiences in software engineering, structured analysis, agile working and writing comprehensive English reports and guides. This is supplemented with new and extended technological knowhow. Just to mention only some of them; Spring, Spring Dynamic Modules, OSGi, Maven and of course Eclipse RCP.

After working with them and using all the helpful tools, I cannot imagine how to work without them anymore.

Thanks

Therefore, I want to thank Wolfgang Giersche and René Eggenschwiler for their trust and support during the whole thesis. The opportunities to discuss and ask to very time, was, in my opinion, very precious. I want also to thank Florian Huonder for his willingness to give us every time support and feedback. And finally a great thank to Ylli for his great engagement and the fantastic teamwork.

13.2 Ylli Sylejmani

Motivation

Already during the term thesis Tobias Forster and I had discussions with René Eggenschwiler about a bachelor thesis in the HERAS^{AF} project. Having made great experiences in the last thesis, it was obvious for us to continue working with the same people in the same project. Also, the ideas for the topic of a bachelor thesis were very interesting. We agreed to choose the development of a reference architecture with Eclipse RCP and OSGi, although we had no experiences with these technologies. It was a great chance to work in a demanding project and be able to use modern software engineering tools and principles. An additional motivation was the fact, that the developed reference architecture will be used in further projects.

Challenges

The familiarization with the new technologies was one of the main challenges. It required a lot of research effort and the implementation of various example applications. Like this, we were able to gather the required know-how and define the requirements for the reference architecture.

Another huge challenge was the collection of useful information and best practices. Especially for the development of the automatic build environment, it was really difficult to find sufficient information about the cooperation of the different tools we were using. The Eclipse RCP and OSGi characteristics further raised the complexity and the configuration effort of the whole buildup. Thus, the development of the automatic build process also was one of the great challenges during this bachelor thesis.

Experiences

Generally, I could gain a lot of technical experiences with Eclipse RCP, OSGi and Spring Dynamic Modules. I really like the concept of these technologies and appreciate the simplicity and flexibility they provide in practice. The integration of

Spring for the wiring of RCP applications was one of the things that impressed me most during the development of the reference architecture.

Another great experience was the possibility to use newest technologies and the proof-of-concept for different variations of solutions. Beside the technologies, I could also gain further know-how in software engineering, agile development and the usage of supporting tools, such as JIRA, Mylyn, Wiki and others. Additionally, I could use various advanced design patterns, like e.g. extension interface, the whiteboard pattern and the like, in practice.

Thanks

I want to thank Wolfgang Giersche and René Eggenschwiler for their assistance and collaboration throughout this bachelor thesis. I appreciate it, that we had the opportunity to discuss and bring ideas at any time. I also want to thank Florian Huonder for taking his time and being willing to support us and give us feedback at all times. Last but not least, I want to thank Tobias Forster for the great teamwork and his motivation.

14 Glossary

API — The **application programming interface** is a set of functions, routines, classes and interfaces provided by libraries, frameworks or operating systems in order to support the development of applications with specific features.

Branding plug-in — The **branding plug-in** is the one, which contains the *.product* file. With this file several branding options can be set like the dependent plug-ins, a startup configuration, launching parameters, a splash screen and further more.

Bundle — A **bundle** is the smallest delivery unit in an OSGi environment. It is a normal Java Archive, or JAR for short, extended with a more detailed *Manifest.mf* file, describing the bundle for the OSGi framework.

CVS — CVS, also known as **Concurrent Versions System**, is a free software revision control system. Version control system software keeps track of all work and all changes in a set of files, and allows several developers (potentially widely separated in space and/or time) to collaborate.

Delta Package — The **delta package** is an extension of the Eclipse Rich Client Platform to allow the generation of products not only for the current platform but also for Linux, Unix, Macintosh and further more. This is required, because SWT and other parts use operating system dependent libraries.

DRY — DRY stands for **Don't repeat yourself** and is a process and programming philosophy with the aim to reduce repetition and duplication.

Eclipse — **Eclipse** is an integrated development environment for Java and further programming languages. It is developed and maintained by the correspondent foundation.

IDE — IDE is an acronym for **Integrated Development Environment**, which is a software application that provides comprehensive facilities to software developers. Typically an IDE consists of source code editors, compilers and interpreters, build automation tools and debugger.

IoC — The **Inversion of Control** is an abstract software engineering principle, describing the aspect of architecture designs in which the flow of control is inverted. Although a certain piece of code is invoked and gets the required parameters, the time when this happens is out of its control. Often it is also known as the *Hollywood Principle*.

JVM — The **Java Virtual Machine** is a software program that uses a virtual machine model for the execution of Java programs. The model is a kind of computer intermediate language also known as the Java bytecode. The advantage of such a virtual machine is the possibility to compile a program only once and to run it everywhere.

KISS — KISS stands for **Keep it simple, stupid.** It is a design philosophy with the main

goal of software simplicity through the avoidance of unnecessary complexity.

Mock

A **mock object** is used in the object-oriented programming to simulate the behavior of real objects in a controlled way. Such objects are typically created through software developer to test the behavior of limited amount of classes.

MVC

MVC stands for **Model-View-Controller**, which is an architectural pattern used in software engineering. Its main purpose is the isolation of the business logic from the user interface dependent implementations, where the exchange of selected parts and their modifications are easier to achieve.

OSGi

OSGi, formerly known as Open Service Gateway initiative, is a standards organization and a Java-based service platform that can be remotely managed.

The core part of the specifications is a framework that defines an application life cycle management model, a service registry, an Execution environment and Modules.

PDE

The **Plug-in Development Environment** is a subproject of the Eclipse Platform. It provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features and further products.

PDE Build

The **PDE Build** is a facility provided by the Eclipse's PDE subproject to allow the automation of the plug-in build process.

Plug-in

A **plug-in** is an on-demand extension of an existing application or service platform, which provides a specific functionality. One of the most popular examples of a plug-in system is the Eclipse platform, which can be extended with several plug-ins.

POJO

POJO is an acronym for **Plain Old Java Object**. It is used to describe an ordinary and simple Java object without special library and framework dependencies.

POM

The **Project Object Model** is part of the Maven providing all configurations for a single project. This includes in general the project name, used properties and dependencies to other projects. Also the extension of the build process through additional plug-ins and the integration of further repository can be done in it.

Product

A **product** within the meaning of the Eclipse PDE Build is a deliverable and executable program or program element. This can be either a whole application or a simple plug-ins or features for the extension of an existing application.

RCP

The **Rich Client Platform** is software consisting of several components like a core, and standard building framework, a portable widget toolkit, file and text handling as well as a workbench management. It allows programmers to build their own applications on existing platforms. Instead of having to write a complete application from scratch, they can benefit from proven and tested features of the framework provided by the platform. Building on a platform facilitates faster application development and integration, while the cross-platform burden is

taken on by the platform developers.

RUP — RUP is an acronym and stands for **Rational Unified Process** and is an interactive software development process framework. It is not a single concrete prescriptive process, but rather a framework which must be adapted for its use.

Scrum — **Scrum** is an interactive and incremental process for managing complex work, like a software development project. It is commonly used with an agile software development.

SOA — SOA stands for **Service-oriented Architecture** whose purpose is to provide methods for the software system development and integration of interoperable but lose-coupled functional packages, called services. These services can be located on the same platform as well as on totally different one.

Spring — The **Spring Framework** is an open source application framework for the Java platform and the .NET Framework.

Spring DM — The **Spring Dynamic Modules** is an extension of the existing Spring Framework for supporting simplifications and utilities based on an OSGi environment.

SVN — SVN, also known as **Subversion**, is a version control system. It is used to maintain current and historical versions of files such as source code, web pages, and documentation.

UI — The **user interface** is that part of an application which is used for the interaction with the user. This covers the visualization of information as well as their manipulation.

15 Bibliography

15.1 Specifications and Standards

[ISO9126] International Organization for Standardization (ISO)

ISO/IEC 9126-1:2001
July 29, 2008

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749

[OSGiCore] OSGi Service Platform

Core Specification
Release 4 Version 4.1
April 2007

http://www.osgi.org/download/r4v41/r4.core.pdf

[OSGiComp] OSGi Service Platform

Compendium Specification

Release 4 Version 4.1
April 2007

http://www.osgi.org/download/r4v41/r4.cmpn.pdf

[SpringDM] Spring Dynamic Modules

Reference Guide
Adrian M. Colyer, Hal Hildebrand, Costin Leau, Andy Piper
Version 1.1.3

http://static.springframework.org/osgi/docs/1.1.3/reference/html/

[SpringRef] The Spring Framework

Reference Documentation
Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma
Version 2.5.x

http://static.springframework.org/spring/docs/2.5.x/reference/

15.2 Books

[Buschmann 1996] Pattern-Oriented Software Architecture

A System of Patterns
1st edition, July 1996
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad
Wiley & Sons

ISBN: 0-471-95869-7

[Daum2008] Rich-Client-Entwicklung mit Eclipse 3.3

Anwendungen entwickeln mit Eclipse RCP, SWT, Forms, GEF, BIRT, JPA, u.a.m.
3. überarbeitete und erweiterte Auflage, 2008
Berhold Daum
dpunkt.verlag

ISBN: 978-3-89864-503-4

[Daum2008] Rich-Client-Entwicklung mit Eclipse 3.3

Update auf Eclipse 3.4
Berhold Daum
dpunkt.verlag

http://www.dpunkt.de/leseproben/2668/Update_auf_Eclipse_3.4.pdf

[GoF1995] Design Patterns

Elements of Reusable Object-Oriented Software
1st edition, December 1995
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Prentice Hall

ISBN: 978-0-20163-361-0

[Matzke2005] Ant

Eine praktische Einführung in das Java-Build-Tool
2., überarbeitete und aktualisierte Auflage, 2005
Bernd Matzke
dpunkt.verlag

ISBN: 3-89864-327-1

[Moore04] Eclipse Development

Using the Graphical Editing Framework and the Eclipse Modeling Framework
Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden
First Edition, February 2004
IBM Redbooks

ISBN: 978-0738453163
http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf

[Wolff2007] Spring 2

Framework für die Java Entwicklung
2. aktualisierte und erweiterte Auflage, 2007
Eberhard Wolff
dpunkt.verlag

ISBN: 978-3-89864-465-5

[Wütherich 2008] Die OSGi Service Platform

Eine Einführung mit Eclipse Equinox
1. Auflage, 2008
Gerd Wütherich, Nils Hartmann, Bernd Kolb, Matthias Lübken
dpunkt.verlag

ISBN: 978-3-89864-457-0

15.3 Websites

15.3.1 Concepts

[Extension Interface06] Java Patterns

Extension Interface
June 28, 2006

http://wiki.hsr.ch/APF/files/ExtensionInterface.pdf

[Kriens04] OSGi Alliance

Listeners Considered Harmful: The "Whiteboard" Pattern
Technical Whitepaper
Revision 2.0
Peter Kriens, BJ Hargrave
August 17, 2004

http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

[MVC07] Java BluePrints

Model-View-Controller
A Pattern Description
December 18, 2007

http://java.sun.com/blueprints/patterns/MVC-detailed.html

[Scrum] Scrum Alliance

Scrum: A team-based framework to develop complex systems and products.

http://www.scrumalliance.org/

[Smells] java.net

Smells To Refactorings
Smells Within Classes and Smells Between Classes

http://wiki.java.net/bin/view/People/SmellsToRefactorings

[SUNDesktop] SUN Developer Network (SDN)

Java SE Desktop Overview

http://java.sun.com/javase/technologies/desktop/

15.3.2 Automatic Build & Development Environment

[Barchfeld05] Eclipse

Build and Test Automation for plug-ins and features
By Markus Barchfeld, Zuehlke Engineering
May 29, 2005

http://www.eclipse.org/articles/Article-PDE-Automation/automation.html

[Coenradie08] Gridshore

Using maven to create an osgi bundle (osgi felix sample)
Jettro Coenradie
February 13, 2008

http://www.gridshore.nl/2008/02/13/using-maven-to-create-an-osgi-bundle-osgi-felix-sample-step-2/

[Eakle05] Javalobby

Using Ant as a Text Substitution Preprocessor
Pete Eakle
April 24, 2005

http://www.javalobby.org/articles/ant-preprocessor/?source=archives

[GoogleSpring OSGi] Google Groups

Spring and OSGi
Setup of a Target Platform

http://groups.google.com/group/spring-osgi/web/org-springframework-osgi-eclipse-target

[Kankanamge 08] Charitha Kankanamge's blog

How to make an OSGI bundle using maven bundle plugin
Charitha Kankanamge
October 12, 2008

http://charithaka.blogspot.com/2008/10/how-to-make-osgi-bundle-using-maven.html

[Lötscher06] Automatisiertes Release-Building von Eclipse RCP Applikationen

October 2006
Stephan Lötscher & Edwin Steiner

http://www.inventage.com/tl_files/inventage/pdfs/chopen_2006_10_10.pdf

[MavenLife cycle] Apache Maven Project

Introduction to the Build Lifecycle
Latest Publish

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[Panier07] IT-Republik - Die RCP-Anwendung am laufenden Band

Build-Automatisierung für RCP-Projekte
Karsten Panier and Lars Gehrken
September 2007

http://it-republik.de/jaxenter/artikel/Die-RCP-Anwendung-am-laufenden-Band-1306.html

[PluginDev Guide] Eclipse Help

Platform Plug-in Developer Guide
Reference

http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/ overview-platform.html

[Schadow08] Dominik Schadow's blog

Eclipse RCP Cross Platform Builds
Dominik Schadow
September 21, 2008

http://blog.trivadis.com/blogs/dominikschadow/archive/2008/09/21/cross-platform-builds-mit-der-target-platform.aspx

[Schirmer] Eclipse Magazin

Better Build My Plug-in
RCP-Anwendungen mit Maven erstellen
Uwe Schirmer

http://www.zuehlke.com/fileadmin/pdf/fachartikel/131_eclipse_magazin_ums_my_plugin.pdf

[Schor08] UIMA

Building Eclipse Plugins with maven-bundle-plugin and friends
Some issues solved
Marshall Schor
April 14, 2008

http://cwiki.apache.org/UIMA/building-eclipse-plugins-with-maven-bundle-plugin-and-friends.html

[Vogel09PDE] Vogella.de

Eclipse Headless build with PDE Build - Tutorial
Lars Vogel and Dominik Zapf
January 11, 2009

http://www.vogella.de/articles/EclipsePDEBuild/article.html

[Zapletal08] knol

Automating Eclipse PDE build
Lukáš Zapletal
October 03, 2008

http://knol.google.com/k/luk-zapletal/automating-eclipse-pde-build/1as80wv4bdzca/4#

15.3.3 OSGi

[Aniszczyk07] IBM – Explore Eclipse's OSGi console

Use and extend the console that drives Eclipse
Chris Aniszczyk
January 30, 2007

http://www.ibm.com/developerworks/library/os-ecl-osgiconsole/index.html

[Hargrave06] OSGi Alliance Forum

split package version numbers
BJ Hargrave
March 31, 2006

https://mail.osgi.org/pipermail/osgi-dev/2006-March/000063.html

[Hathi2008] IBM – OSGi and Spring

Build and deploy OSGi bundles using Apache Felix
Rajeev Hathi and Naveen Balani
October 30, 2008

http://www.ibm.com/developerworks/opensource/library/ws-osgi-spring1/index.html

[Kriens06Apr] OSGi Alliance

Misconceptions about OSGi Headers
Peter Kriens
April 3, 2006

http://www.osgi.org/blog/2006/04/misconceptions-about-osgi-headers.html

[Kriens06Oct] OSGi Alliance

JSR 277 Review
Peter Kriens
October 19, 2006

http://www.osgi.org/blog/2006/10/jsr-277-review.html

[Walls08] SpringOne Americas

Compass: A thorn in my OSGi side
Craig Walls
August 4, 2008

http://americas.springone.com/blog/craig_walls/2008/08/compass_a_thorn_in_my_osgi_side.html

[Walls09] Javalobby

A Dozen OSGi Myths and Misconceptions
Craig Walls
April 14, 2009

http://java.dzone.com/articles/dozen-osgi-myths-and

15.3.4 Eclipse RCP

[Burnette06] Eclipse Corner Article

Rich Client Tutorial
Ed Burnette
February 6, 2006

http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html
http://www.eclipse.org/articles/Article-RCP-2/tutorial2.html
http://www.eclipse.org/articles/Article-RCP-3/tutorial3.html

[Ebert09] Eclipse RCP

Ralf Ebert
2009

http://www.ralfebert.de/rcpbuch/

[EclipseHelp] Eclipse

Eclipse Ganymede Documentation

http://help.eclipse.org/ganymede/index.jsp

[EclipseWiki08] Eclipse Wiki

Platform UI Error Handling
July 21, 2008

http://wiki.eclipse.org/Platform_UI_Error_Handling

[Hennig08] Java Spektrum

Einführung in den "Extension Point"-Mechanismus von Eclipse
Manfred Hennig, Heiko Seeberger
2008

http://www.sigs.de/publications/js/2008/01/hennig_seeberger_JS_01_08.pdf

[Paulin] RCP Quickstart

Patrick Paulin

http://rcpquickstart.com/

[Tödter07] Eclipse RCP

Tips & Tricks
Kai Tödter
2007

http://www.parleys.com/.../Home#slide=1;title=Eclipse%20RCP%20Tips%20and%20Tricks;talk=8127

[Vogel09] Vogella.de

Eclipse RCP
Tutorial with Eclipse 3.4
Lars Vogel
March 22, 2009

http://www.vogella.de/articles/RichClientPlatform/article.html

15.3.4.1 Drag and Drop

[Arthorne03] Eclipse Corner Article

Drag and Drop in the Eclipse UI
John Arthorne
August 25, 2003

http://www.eclipse.org/articles/Article-Workbench-DND/drag_drop.html

[Irvine03] Eclipse Corner Article

Drag and Drop
Adding Drag and Drop to an SWT Application
Veronika Irvine
August 25, 2003 (revised November 2, 2005)

http://www.eclipse.org/swt/snippets/

15.3.4.2 Common Navigator Framework

[CNFProg Guide] Eclipse Help

Platform Plug-in Developer Guide
Common Navigator Framework

http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/cnf.htm

[Elder06] Digital Paper Napkin Blog

Building a Common Navigator based viewer
Michael D. Elder
May 20, 2006 - June 18, 2006

http://scribbledideas.blogspot.com/2006/05/building-common-navigator-based-viewer.html
http://scribbledideas.blogspot.com/2006/05/building-common-navigator-based-viewer_22.html
http://scribbledideas.blogspot.com/2006/06/building-common-navigator-based-viewer.html
http://scribbledideas.blogspot.com/2006/06/building-common-navigator-based-viewer_18.html
http://scribbledideas.blogspot.com/2006/06/building-common-navigator-_115067357450703178.html

15.3.4.3 Data Binding

[DataBinding] Eclipse Wiki

JFace Data Binding
Concepts, Tutorials & Presentations

http://wiki.eclipse.org/index.php/JFace_Data_Binding

[Delap06] IBM – Understanding JFace data binding in Eclipse

The pros and cons of data binding
Scott Delap
September 26, 2006

http://www.ibm.com/developerworks/opensource/library/os-ecl-jfacedb1/

[Mittermeier07] IT-Republik - Die RCP-Anwendung am laufenden Band

Eclipse DataBinding für die Kommunikation zwischen Modell und GUI
Ludwig Mittermeier
November 2007

http://it-republik.de/jaxenter/artikel/Eclipse-DataBinding-fuer-die-Kommunikation-zwischen-Modell-und-GUI-1353.html

[Vogel09Data Binding] Vogella.de

Eclipse Databinding with Eclipse RCP applications
Tutorial
Lars Vogel
March 23, 2009

http://www.vogella.de/articles/EclipseDataBinding/article.html

15.3.4.4 Menu- & Toolbar

[EclipseWiki Menu] Eclipse Wiki

Menu Contributions

http://wiki.eclipse.org/index.php/Menu_Contributions

[Prakash09-1] Eclipse Tips

Commands Part 1: Actions Vs Commands
Prakash G.R.
January 2, 2009

http://blog.eclipse-tips.com/2009/01/commands-part-1-actions-vs-commands.html

[Prakash09-2] Eclipse Tips

Commands Part 2: Selection and Enablement of IHandlers
Prakash G.R.
January 5, 2009

http://blog.eclipse-tips.com/2009/01/commands-part-2-selection-and.html

[Voigt08] IBM – Configuring and adding menu items in Eclipse

Use V3.3's new menu mechanism to accelerate plug-in and RCP development
Karsten Voigt
February 19, 2008

http://www.ibm.com/developerworks/library/os-eclipse-3.3menu/index.html

15.3.4.5 Problems View

[Prakash08] Eclipse Tips

Creating a custom Marker View
Prakash G.R.
November 20, 2008

http://blog.eclipse-tips.com/2008/11/creating-custom-marker-view.html

[WBResource Support] Eclipse Help

Platform Plug-in Developer Guide
Advanced Workbench concepts » Workbench resource support

http://help.eclipse.org/ganymede/index.jsp?topic=.../guide/workbench_resources.htm

15.3.4.6 SWT

[EclipseZone Subclassing] EclipseZone

Extending and adding a SWT Button as a new control in Visual Editor

http://www.eclipsezone.com/eclipse/forums/t88689.html

[Northover01] Eclipse Corner Article

Creating Your Own Widgets using SWT
Steve Northover & Carolyn MacLeod
March 22, 2001

http://www.eclipse.org/articles/Article-Writing Your Own Widget/Writing Your Own Widget.htm

[SWTSnippets] Eclipse - SWT: The Standard Widget Toolkit

SWT Snippets

http://www.eclipse.org/swt/snippets/

[SWTSubclass] Eclipse - SWT: The Standard Widget Toolkit

FAQ: Subclassing Widgets
Why can't I subclass SWT widgets like Button and Table?

http://www.eclipse.org/swt/faq.php#subclassing

15.3.4.7 Eclipse Modeling Framework (EMF)

[EMF] Eclipse Projects

Eclipse Modeling Framework (EMF)

http://www.eclipse.org/modeling/emf/

[EMFProg Guide] Eclipse Help

EMF Developer's Guide
Programmer's Guide

June 16, 2005

http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html

[Griffin02] Eclipse Corner Article

Using EMF
Catherine Griffin
December 9, 2002

http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html

[Powell04-1] IBM – Model with the Eclipse Modeling Framework

Part 1: Create UML models and generate code
Adrian Powell
April 15, 2004

http://www.ibm.com/developerworks/opensource/library/os-ecemf1/

[Powell04-2] IBM – Model with the Eclipse Modeling Framework

Part 2: Generate code with Eclipse's Java Emitter Templates
Adrian Powell
April 27, 2004

http://www.ibm.com/developerworks/opensource/library/os-ecemf2/

15.3.4.8 Graphical Editing Framework (GEF)

[Aniszczyk05] Eclipse Corner Article

Using GEF with EMF
Chris Aniszczyk
June 8, 2005

http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html

[Bordeau03] Eclipse Corner Article

Using Native Drag and Drop with GEF
Eric Bordeau
August 25, 2003

http://www.eclipse.org/articles/Article-GEF-dnd/GEF-dnd.html

[Ermel08-01] TU Berlin » Visuelle-Sprachen-Projekt

Einführung in das Graphische Editor Framework (GEF) von Eclipse
Claudia Ermel, Leen Lambers, Tony Modica, EnricoBiermann
April 26, 2008

http://tfs.cs.tu-berlin.de/vila/www_ss08/folien/4-GEF.pdf

[Ermel08-02] TU Berlin » Visuelle-Sprachen-Projekt

Graphische Editor Framework (GEF) – Teil 2
Claudia Ermel, Leen Lambers, Tony Modica, Enrico Biermann
April 28, 2008

http://tfs.cs.tu-berlin.de/vila/www_ss08/folien/5-GEF_Teil2.pdf

[Ermel08-03] TU Berlin » Visuelle-Sprachen-Projekt

GEF-Editor Features: Anchor Points, Actions, DirectEdit
Claudia Ermel, Leen Lambers, Tony Modica, Enrico Biermann
May 8, 2008

http://tfs.cs.tu-berlin.de/vila/www_ss08/folien/7-GEF_Teil3.pdf

[GEF] Eclipse Projects

Graphical Editing Framework (GEF)

http://www.eclipse.org/gef/

[GEFDev Guide] Eclipse Help

GEF and Draw2d Plug-in Developer Guide
Programmer's Guide

http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html

[Hudson04] EclipseCon 2004

The Graphical Editing Framework
Randy Hudson
February 2, 2004

http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/47_Hudson.pdf

[Hudson06] IBM – Graphical Editing Framework

Create an Eclipse-based application using the Graphical Editing Framework
Randy Hudson
June 27, 2006 (Update)

http://www.ibm.com/developerworks/opensource/library/os-gef/

[Hudson07] IBM – Graphical Editing Framework

Create an Eclipse-based application using the Graphical Editing Framework
Randy Hudson, Chris Aniszczyk
March 27, 2007

http://www.ibm.com/developerworks/library/os-eclipse-gef11/

[IEditorInput08] A Lapse in Reason

Using IEditorInput
April 7, 2008

http://www.pompo.us/2008/04/07/using-ieditorinput/

[Lee03] Eclipse Corner Article

Display a UML Diagram using Draw2D
Daniel Lee
August 25, 2003

http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

[Lemaigre09] Eclipse Wiki

GEF Description

Régis Lemaigre

http://eclipsewiki.editme.com/GefDescription

[Majewski04] Eclipse Corner Article

A Shape Diagram Editor
Bo Majewski
December 8, 2004

http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html

[Mammana08] GEF (Graphical Editing Framework)

Tutorial
Jean-Charles Mammana, Romain Meson, Jonathan Gramain
April 26, 2008

http://www.psykokwak.com/blog/images/gef/GEF_Tutorial.pdf

15.3.4.9 Graphical Modeling Framework (GMF)

[Aniszczyk06] IBM – GMF

Learn Eclipse GMF in 15 minutes
Chris Aniszczyk
September 12, 2006

http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

[GMF] Eclipse Projects

Graphical Modeling Framework (GMF)

http://www.eclipse.org/modeling/gmf/

[GMFGraph Hints] Eclipse Wiki

GMFGraph Hints / GMF GenModel Hints

http://wiki.eclipse.org/GMFGraph_Hints / http://wiki.eclipse.org/GMF_GenModel_Hints

[GMFTips] Eclipse Wiki

GMF Tips

http://wiki.eclipse.org/GMF_Tips

[GMFTrouble Guide] Eclipse Wiki

GMF Troubleshooting Guide

http://wiki.eclipse.org/GMF_Troubleshooting_Guide

[GMFTutorial] Eclipse Wiki

GMF Tutorial

http://wiki.eclipse.org/index.php/GMF_Tutorial

[Kolb06] Graphical Modeling Framework

An Introduction
Bernd Kolb, Sven Efftinge, Markus Voelter, Arno Haase
December 17, 2006

http://www.voelter.de/data/articles/ix-gmf2.pdf

[Kulekova06] Graphische Editoren mit Eclipse

EMF, GEF & GMF
Krasimira Kulekova, Christian Köhler (Technische Universität Berlin)
February 5, 2006

http://user.cs.tu-berlin.de/~vila/www_ws05.sav/folien/Vortrag-emf-gef-gmf.pdf

[Plante06] Eclipse Corner Article

Introducing the GMF Runtime
Frederic Plante
January 16, 2006

http://www.eclipse.org/articles/Article-Introducing-GMF/article.html

[Vanden-houten08] Modellgetriebene Entwicklung mit Eclipse GMF

Wie programmiert man einen graphischen Editor ohne eine Zeile Quellcode?
Ralf Vandenhouten
November 26, 2008

http://www.tm.tfh-wildau.de/index.php?option=com_content&task=view&id=90&Itemid=29
http://vandenhouten.de/media/GMF-Step-By-Step.pdf

15.3.4.10 Text Editor

[Eicher05] Entwickler.de

Raffinierte Rezepte: Neue Funktionen für Ihren Eclipse 3.0-Texteditor
Tom Eicher, Markus Keller, Christof Marti
August, 2005

http://it-republik.de/zonen/portale/psecom,id,101,online,647,neu,1.html

[Eicher06] Eclipse CON 2006

Text Editor Recipes
Tom Eicher
March, 2006

http://www.eclipse.org/eclipse/platform-text/eclipseCon/2006/texteditorrecipes.pdf

[Zoio06] Realsolve Developer Corner

Building an Eclipse Text Editor with JFace Text
Phil Zoio
April, 2006

http://www.realsolve.co.uk/site/tech/jface-text.php

15.3.4.11 Views

[Springgay01] Eclipse Corner Article

Creating an Eclipse View
Dave Springgay
November 2, 2001

http://www.eclipse.org/articles/viewArticle/ViewArticle2.html

15.3.5 Spring Dynamic Modules

[Gleichmann08] brain driven development

Setting up Spring Dynamic Modules (OSGi) with Eclipse
A step-by-step tutorial
Mario Gleichmann
March 24, 2008

http://gleichmann.wordpress.com/2008/03/24/setting-up-spring-dynamic-modules-osgi-with-eclipse-a-step-by-step-tutorial/

[Hathi09] IBM - OSGi and Spring

Build and deploy OSGi as Spring bundles using Felix
Rajeev Hathi, Naveen Balani
March 30, 2009

http://www.ibm.com/developerworks/opensource/library/ws-osgi-spring2/index.html

[Kriens07] OSGi Alliance

Spring and OSGi: Jumping Beans
Peter Kriens
January, 9, 2007

http://www.osgi.org/blog/2007/01/spring-and-osgi-jumping-beans.html

[Lippert08] Martin Lippert Blog

Springframework Utilities
Martin Lippert
October 4, 2008

http://martinlippert.blogspot.com/

[Patil08] Java World

Introduction to Spring Dynamic Modules
Build a service-oriented application using Spring and OSGi
Sunil Patil
April 22, 2008

http://www.javaworld.com/javaworld/jw-04-2008/jw-04-osgi2.html?page=1

[Thurow08] IT-Republik - Enterprise Eclipse

Spring trifft auf OSGi und RCP
Alexander Thurow, Heiko Seeberger
25. August 2008
Only short description and source code

http://it-republik.de/jaxenter/eclipse-magazin-ausgaben/Enterprise-Eclipse-000268.html

15.3.6 Logging

[Franey05] Eclipse Zone

Universal Logger Plug-ins for RCP Applications
John Franey
November 2005

http://www.eclipsezone.com/articles/franey-logging/

[Marques04] IBM - Plugging in a logging framework for Eclipse plug-ins

Two approaches to improve your Eclipse logging experience
Manoel Marques
September 27, 2004

http://www.ibm.com/developerworks/library/os-eclog/

15.3.7 Testing

[TestNGDoc] TestNG

TestNG testing framework documentation

http://testng.org/doc

15.4 Other resources

[ApacheAnt] http://ant.apache.org/

[ApacheFelix] http://felix.apache.org/

[Bamboo] http://www.atlassian.com/software/bamboo/

[EclEmma] http://www.eclemma.org/

[EclipseIAM] http://www.eclipse.org/iam/

[EclipseMetrics] http://www.stateofflow.com/

[EclipseRCP] http://www.eclipse.org/rcp/

[Equinox] http://www.eclipse.org/equinox/

[JUnit] http://www.junit.org/

[Log4J] http://logging.apache.org/log4j/

[Maven] http://maven.apache.org/

[MVNAntRun] http://maven.apache.org/plugins/maven-antrun-plugin/

[MVNBundle] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

[MVNCompiler] http://maven.apache.org/plugins/maven-compiler-plugin/

[MVNJavadoc] http://maven.apache.org/plugins/maven-javadoc-plugin/

[MVNOSGi] http://mavenosgiplugin.berlios.de/

[MVNPDE] http://mojo.codehaus.org/pde-maven-plugin/

[MVNResources] http://maven.apache.org/plugins/maven-resources-plugin/

[MVNSource] http://maven.apache.org/plugins/maven-source-plugin/

[Mylyn] http://www.eclipse.org/mylyn/

[Q4E] http://q4e.googlecode.com/

[SLF4J] http://www.slf4j.org/

[Spring] http://www.springframework.org/

[SpringIDE] http://dist.springframework.org/release/IDE

[SpringDM] http://www.springframework.org/osgi/

[Subversive] http://www.eclipse.org/subversive/

[TestNG] http://testng.org/

[Tycho] http://docs.codehaus.org/display/M2ECLIPSE/Tycho+project+overview

[WickedShell] http://www.wickedshell.net/

[wikiTestNG] http://en.wikipedia.org/wiki/TestNG

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HERAS-AF

Appendix B: Tutorials

16 Create a simple OSGi based application

Overview

In this tutorial we show you how to create an OSGi based application step by step. This means we build the application bundle by bundle with the ulterior motive to focus each bundle on its core concerns. We implement the application in the following steps:

- Creating a parent project
- Creating an OSGi service
- Creating an OSGi client
- Creating unit tests
- Creating integration tests

For an easier build and development, we use the Maven build tool, which allows the generation of manifest files and loads the required bundles respectively libraries out of the configured repositories.

Additionally we use the Spring Dynamic Modules, which supports the declarative import and export of services for OSGi bundles. Without these bundles, we would have to implement the service registration and allocation on our own.

Conditions

A requirement for a successful implementation of this tutorial is the existence of an infrastructure, how it is described in chapter 8.1, expect for the Automatic Build out of subchapter 8.1.5 we integrate in this tutorial. It's up to you to decide if you want to use the same infrastructure. You can also use substitutes for some tools or different versions, but we cannot guarantee the success.

Additionally we suppose you have some experiences with OSGi, Spring DM and Maven. Otherwise it is hard to understand the following tutorial.

Notice

Note we give in this tutorial only a small amount of descriptions and explanations about the concepts and decisions. They are all described in detail in the chapters 6and 8.

16.1 Create a parent project

Intention

In a first step we create a parent project. The main purpose of this project is to configure dependencies and options in Maven which should be applied for all of our following projects respectively bundles. It allows also an easy build of all of our projects we will create now.

Tutorial

1. After you opened your Eclipse choose in the menu *File* > *New* > *Other…*
2. In the dialog which opens, select the item *Maven 2 Project* > *Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name "org.herasaf.tutorial" in the field *project name*.

4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. Now we created our parent project. Since this project contains only Maven configurations, you can delete the *src* and *target* folder.
8. After this step you should only have the *Maven Classpath Container*, the *JRE System Library* and the *pom.xml* file in our project.
9. Open now the *pom.xml* file and change to the source view.
10. Override its content with the following snippet. We define default properties which are used for the maven bundle plug-in, add the Spring DM bundles as shared dependencies, configure the used plug-ins especially the maven bundle plug-in, and add repositories to get the Spring DM bundles.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.herasaf.tutorial</groupId>
  <artifactId>tutorial-osgi</artifactId>
  <packaging>pom</packaging>
  <version>1.0.0</version>
  <name>OSGi Tutorial (Parent)</name>

  <properties>
    <manifest.symbolicname>
      ${project.groupId}.${project.artifactId}
    </manifest.symbolicname>

    <manifest.private></manifest.private>
    <manifest.export></manifest.export>
    <manifest.import>
      !${service.export},
      *;resolution:=optional
    </manifest.import>
    <manifest.activator></manifest.activator>
    <manifest.require></manifest.require>
    <manifest.resource></manifest.resource>
    <manifest.classpath>.</manifest.classpath>
    <manifest.environment>J2SE-1.5</manifest.environment>
    <manifest.activationPolicy></manifest.activationPolicy>

    <!-- Other properties -->
    <spring.osgi.version>1.1.3</spring.osgi.version>
  </properties>

  <modules>
  </modules>

  <!-- Shared dependencies -->
  <dependencies>

    <dependency>
```

```
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-annotation</artifactId>
      <version>${spring.osgi.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-core</artifactId>
      <version>${spring.osgi.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-extender</artifactId>
      <version>${spring.osgi.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-io</artifactId>
      <version>${spring.osgi.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-mock</artifactId>
      <version>${spring.osgi.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-test</artifactId>
      <version>${spring.osgi.version}</version>
      <scope>test</scope>
    </dependency>

  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-javadoc-plugin</artifactId>
        <executions>
```

```
          <execution>
            <phase>package</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <aggregate>false</aggregate>
          <quiet>true</quiet>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.0.0</version>
        <extensions>true</extensions>
        <configuration>
          <manifestLocation>META-INF</manifestLocation>
          <instructions>
           <Bundle-Name>
             ${artifactId}</Bundle-Name>
           <Bundle-SymbolicName>
             ${manifest.symbolicname}
           </Bundle-SymbolicName>
           <Import-Package>
             ${manifest.import}
           </Import-Package>
           <Export-Package>
             ${manifest.export}
           </Export-Package>
           <Private-Package>
             ${manifest.private}
           </Private-Package>
           <Bundle-Activator>
             ${manifest.activator}
           </Bundle-Activator>
           <Require-Bundle>
             ${manifest.require}
           </Require-Bundle>
           <Include-Resource>
             ${manifest.resource}
           </Include-Resource>
           <Bundle-RequiredExecutionEnvironment>
             ${manifest.environment}
           </Bundle-RequiredExecutionEnvironment>
           <Bundle-ActivationPolicy>
             ${manifest.activationPolicy}
           </Bundle-ActivationPolicy>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
      <id>com.springsource.repository.bundles.external</id>
      <name>SpringSource Enterprise Bundle Repository - External
Bundle Releases</name>
      <url>
        http://repository.springsource.com/maven/bundles/
        external
      </url>
    </repository>
```

```
    <repository>
      <id>com.springsource.repository.bundles.release</id>
      <name>SpringSource Enterprise Bundle Repository - Spring-
Source Bundle Releases</name>
      <url>
        http://repository.springsource.com/maven/bundles/
        release
      </url>
    </repository>

    <repository>
      <id>spring-release</id>
      <name>Spring Portfolio Release Repository</name>
      <url>http://maven.springframework.org/release</url>
    </repository>

    <repository>
      <id>spring-external</id>
      <name>Spring Portfolio External Repository</name>
      <url>http://maven.springframework.org/external</url>
    </repository>

    <repository>
      <id>spring-milestone</id>
      <name>Spring Portfolio Milestone Repository</name>
      <url>http://maven.springframework.org/milestone</url>
    </repository>

    <repository>
      <id>spring-ext</id>
      <name>Spring External Dependencies Repository</name>
      <url>
        http://springframework.svn.sourceforge.net/svnroot/
        springframework/repos/repo-ext/
      </url>
    </repository>

    <!-- used when building against Spring snapshots -->
    <repository>
      <id>spring-snapshot</id>
      <name>Spring Portfolio Milestone Repository</name>
      <url>http://maven.springframework.org/snapshot</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>

    <repository>
      <id>i21-s3-osgi-repo</id>
      <name>i21 osgi artifacts repo</name>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <url>http://maven.springframework.org/osgi</url>
    </repository>

  </repositories>

</project>
```

Listing 233: The pom.xml of the org.herasaf.tutorial project

11. After the configuration of Maven in our parent project, we have nothing

more to do in this project.

16.2 Create an OSGi Service

Intention

Now we want to implement our OSGi service which resolves texts for given keys like the `ResourceBundle` does.

We create in this step two bundles, one for the service declaration and one with the concrete implementation of it. The idea behind this separation is the possibility to provide additional service implementations in a later step or by other developers. If you have the interface and the concrete implementation of your service in the same bundle this possibility is only hard to achieve.

At the moment we only develop one concrete implementation. But feel free to implement additional bundles with other implementations.

Tutorial: Service declaration

1. We begin with the development of the service declaration bundle. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.service” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.service”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. Now we created our service declaration project. It’s a normal Java project with Maven support, but we need a Plug-in Project. So we have to convert it. Select the project in the *Project Explorer* and do a right click on it. Choose *PDE Tools > Convert Projects to Plug-in Projects…*
8. In the dialog which opens, select now our project you want to convert and click *Finish*.
9. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don’t use it.
10. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
11. Override its content with the following snippet. We link it with our parent project to benefit from the common configurations. Additionally, we override the property `<manifest.export>`, because we want to export the interface of our service.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial</groupId>
    <artifactId>tutorial-osgi</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>tutorial-osgi-service</artifactId>
  <name>OSGi Tutorial - Service</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.export>
      org.herasaf.tutorial.service;version=${project.version}
    </manifest.export>
  </properties>
</project>
```

Listing 234: The pom.xml of the org.herasaf.tutorial.service project

12. Additionally we have to add a link from the parent to our service declaration project. Open for that purpose the *pom.xml* file from our parent project.
13. Add the following line between the <modules> tags.
    `<module>../org.herasaf.tutorial.service</module>`
14. Now we can close the *pom.xml* of the parent project and return to the service declaration project.
15. Create in the folder *src/main/java* a new interface `org.herasaf.tutorial.service.TranslationService`.
16. Open the interface and add the following method declaration.
    `public String getTranslation(String key);`
17. That's it, we created our service declaration. Try to build the *org.herasaf.tutorial* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

Tutorial: Service implementation

18. After the creation of the service declaration we can develop the service implementation bundle. Choose in the Eclipse menu *File* > *New* > *Other…*
19. In the dialog which opens, select the item *Maven 2 Project* > *Maven 2 Project Creation Wizard* and click the *Next* button.
20. Now enter the name "org.herasaf.tutorial.service.impl" in the field *project name*.
21. Select the option *Create a simple java project* and click the *Next* button.
22. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values "org.herasaf.tutorial.service.impl".
23. The *Version* field should contain "1.0-SNAPSHOT". Change it to "1.0.0"

and click the *Finish* button.

24. Now we created our service implementation project. We also need to convert it to a Plug-in Project. Select the project in the *Project Explorer* and do a right click on it. Choose *PDE Tools > Convert Projects to Plug-in Projects…*
25. In the dialog which opens, select now our project you want to convert and click *Finish*.
26. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don't use it.
27. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
28. Override its content with the following snippet. We link it also with our parent project to benefit from the common configurations. Additionally, we override properties to generate a correct manifest file. Additionally we add the dependency to the service declaration project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial</groupId>
    <artifactId>tutorial-osgi</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>tutorial-osgi-service-impl</artifactId>
  <name>OSGi Tutorial - Service Impl</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.private>
      org.herasaf.tutorial*impl*
    </manifest.private>

    <manifest.import>
      !${service.export},
      org.herasaf.tutorial.service;version=${project.version},
      *;resolution:=optional
    </manifest.import>

    <manifest.resource>
      src/main/resources/translation.properties,
      META-INF/spring=META-INF/spring
    </manifest.resource>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial</groupId>
      <artifactId>tutorial-osgi-service</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```

Listing 235: The pom.xml of the org.herasaf.tutorial.service.impl project

29. Additionally we have to add a link from the parent to our service implementation project. Open for that purpose the *pom.xml* file from our parent project.
30. Add the following line between the <modules> tags.
`<module>../org.herasaf.tutorial.service.impl</module>`
31. Now we can close the *pom.xml* of the parent project and return to the service implementation project.
32. Before we can develop our service implementation, we have to run a Maven build on this project to generate the manifest file. Otherwise we cannot access the `TranslationService` interface.
Open the console and run the build with the command `mvn org.apache.felix:maven-bundle-plugin:manifest`.
33. When you now check the manifest file you can see the values of the properties we configured in the *pom.xml*. You maybe have to refresh your Project Explorer instead.
34. Now, we implement our service. Create in the folder *src/main/java* a new class `org.herasaf.tutorial.service.impl.TranslationServiceImpl`.
35. Open the class and add the following snippet.

```
public class TranslationServiceImpl implements TranslationService
{

  private ResourceBundle translations;

  public TranslationServiceImpl() {
    this(Locale.getDefault());
  }

  public TranslationServiceImpl(Locale locale) {
    try {
      translations = ResourceBundle.getBundle("translation", lo-
cale);
    } catch (MissingResourceException exc) {
      translations = ResourceBundle.getBundle("translation");
    }
  }

  public String getTranslation(String key) {
    try {
      return translations.getString(key);
    } catch (MissingResourceException exc) {
      return key;
    }
  }
}
```

Listing 236: The implementation of the TranslationServiceImpl

36. After we've implemented the service, we have to create a properties file which contains the keys and texts. Create a new folder with the name *src/main/resources* in the project.
37. Create a new file with the name *translation.properties* and add key-

value pairs of your choice.

38. We've implemented now the translation service but we have to register it in the OSGi service registry. We use the Spring Dynamic Modules to do this.
39. Create a new folder *spring* under the existing folder *META-INF*.
40. Create the file *service.xml* in the new folder and add the following snippet. We use this file for the normal Spring instantiation and dependency injection.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="translation" class="org.herasaf.tutorial.service.
    impl.TranslationServiceImpl" />

</beans>
```

Listing 237: The translation service instantiation in the service.xml

41. Now add an additional file *service-osgi.xml* in the same folder and add the following snippet. We use this file to register the instantiated service.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <osgi:service id="translationOsgi" ref="translation"
   interface="org.herasaf.tutorial.service.TranslationService" />

</beans>
```

Listing 238: The translation service registering in the service-osgi.xml

42. That's it, we created our service implementation. Try to build the *org.herasaf.tutorial* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

16.3 Create an OSGi client

Intention

Until now we implemented a service which can resolve a text for a given key. Now we want to implement the counter piece of the service, its client.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.client” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.client”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. We created our client project as a normal Java project with Maven support. So we also have to convert it to *Project Explorer*. Do a right click on the project and choose *PDE Tools > Convert Projects to Plug-in Projects…*
8. In the dialog which opens, select now our project you want to convert and click *Finish*.
9. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don’t use it.
10. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
11. Override its content with the following snippet. We link it with our parent project to benefit from the common configurations. Additionally, we override the properties to generate a correct manifest file and add the dependency to the service declaration project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial</groupId>
    <artifactId>tutorial-osgi</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>tutorial-osgi-client</artifactId>
  <name>OSGi Tutorial - Client</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.import>
      !${service.export},
      org.herasaf.tutorial.service;version=${project.version},
      *;resolution:=optional
    </manifest.import>

    <manifest.resource>
      META-INF/spring=META-INF/spring
    </manifest.resource>
```

```
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial</groupId>
      <artifactId>tutorial-osgi-service</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>

</project>
```

Listing 239: The pom.xml of the org.herasaf.tutorial.client project

12. Additionally we have to add a link from the parent to our service client project. Open for that purpose the *pom.xml* file from our parent project.
13. Add the following line between the <modules> tags.
    `<module>../org.herasaf.tutorial.client</module>`
14. Now we can close the *pom.xml* of the parent project and return to the client project.
15. Before we can develop our client using the service declaration, we have to run a Maven build on this project to generate the manifest file. Otherwise we cannot access the `TranslationService` interface.
    Open the console and run the build with the command `mvn org.apache.felix:maven-bundle-plugin:manifest`.
16. When you now check the manifest file you can see the values of the properties we configured in the *pom.xml*. You maybe have to refresh your Project Explorer instead.
17. Create now a new class `org.herasaf.tutorial.client.ServiceCaller` in the folder *src/main/java*.
18. Open the class and add the following snippet.

```
public class ServiceCaller {

  private TranslationService service = null;

  public void setService(TranslationService service) {
    this.service = service;
  }

  public void init() {
    execute();
  }

  public void stop() {
    service = null;
  }

  public void execute() {
    if (service != null) {
      String appName = service.getTranslation("appName");
      System.out.println("The name of the application is: " +
appName);
    }
  }
}
```

Listing 240: The implementation of the ServiceCaller

19. We've implemented now the service client, but we have to get the reference to the service from the OSGi service registry. We use also in this case the Spring Dynamic Modules to do this.
20. Create a new folder *spring* under the existing folder *META-INF*.
21. Create the file *client.xml* in the new folder and add the following snippet. We use this file for the normal Spring instantiation and dependency injection.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="serviceCaller" class="org.herasaf.tutorial.client.
    ServiceCaller" init-method="init" destroy-method="stop" />
    <property name="service" ref="translationService" />
  </bean>
</beans>
```

Listing 241: The service caller instantiation in the client.xml

22. Now add an additional file *client-osgi.xml* in the same folder and add the following snippet. We use this file to get the reference of the service.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <osgi:reference id="translationService"
    interface="org.herasaf.tutorial.service.TranslationService"/>

</beans>
```

Listing 242: The translation service reference in the client-osgi.xml

23. That's it, we created our service client. Try to build the *org.herasaf.tutorial* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.
24. Now we can execute our application in Eclipse. Open the *Run Configurations*.
25. Create a new *OSGi Framework* run configuration.
26. Deselect all bundles in the tab *Bundles*.
27. Select now the following bundles:
    tutorial-osgi-client, *tutorial-osgi-service*, *tutorial-osgi-service-impl*, *org.eclipse.osgi*, *org.springframework.bundle.osgi.core*,

org.springframework.bundle.osgi.extender and *org.springframework.bundle.osgi.io*.

28. Click now the button *Add Required Bundles* and run the application.
29. Now you should see an output from the application in the console.

16.4 Create unit tests

Intention

After we developed an OSGi based application with a service and a client, we want to implement unit tests for verification. They should also be integrated in the Maven build process.

Since the plug-ins in Eclipse cannot yet handle the different class paths for the main and test sources, we have to locate the unit tests in a separate Eclipse project.

For the unit test implementation we use TestNG instead of JUnit. It provides a better testing support with a better functional range.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.unittest” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.unittest”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. We created our unit test project as a normal Java project with Maven support. Different than the client and service projects, we don’t need to convert it to a Plug-in Project.
8. Now we are going to implement our unit tests. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
9. Override its content with the following snippet. We link it also with our parent project to benefit from the common configurations. Additionally we add the dependency to the service implementation project and to the testing libraries we are using later.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
```

```
    <groupId>org.herasaf.tutorial</groupId>
    <artifactId>tutorial-osgi</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>tutorial-osgi-unittest</artifactId>
  <name>OSGi Tutorial - Unit Test</name>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial </groupId>
      <artifactId>tutorial-osgi-service-impl</artifactId>
      <version>${project.version}</version>
    </dependency>

    <!-- Testing dependencies -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>2.5.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>5.7</version>
      <classifier>jdk15</classifier>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>2.5.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Listing 243: The pom.xml of the org.herasaf.tutorial.unittest project

10. Additionally we have to add a link from the parent to our unit test project. Open for that purpose the *pom.xml* file from our parent project.
11. Add the following line between the <modules> tags.
    `<module>../org.herasaf.tutorial.unittest</module>`
12. Now we can close the *pom.xml* of the parent project and return to the unit test project.
13. Create now a new source folder with the name *src/test/java*.
14. In this folder, create a new class `org.herasaf.tutorial.unittest.TranslationServiceImplTest`.
15. Open the class and add the following snippet.

```
@ContextConfiguration(locations = { "class-
path:/applicationContext.xml" })
public class TranslationServiceImplTest extends
    AbstractTestNGSpringContextTests {

  @Autowired
```

```
  private TranslationServiceImpl service;

  @Test
  public void testAvailableText() throws Exception {

    assertNotNull(service);

    String key = "appName";
    String translation = "OSGi Tutorial Service";
    assertEquals(translation, service.getTranslation(key));
  }

  @Test
  public void testNotAvailableText() throws Exception {

    assertNotNull(service);

    String key = "Rührei";
    assertEquals(key, service.getTranslation(key));
  }
}
```

Listing 244: The implementation of the TranslationServiceImplTest

16. We've implemented now the unit test cases, but we still have to configure it with the normal Spring dependency injection. So we create a new source folder with the name *src/test/resources*.
17. Now create the file *applicationContext.xml* in this folder.
18. Open the file and add the following code snippet.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/ schema/beans/spring-beans-
2.5.xsd">

  <bean id="service" class="org.herasaf.tutorial.service.impl.
      TranslationServiceImpl" />

</beans>
```

Listing 245: The unit test application context in the applicationContext.xml

19. Now our unit tests are complete. You can run it now in Eclipse as a normal TestNG test or you can try to build the *org.herasaf.tutorial* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

16.5 Create integration tests

Intention

We developed now a unit test for a specific bundle to verify it. But when we want to test the workflows in a bigger application with more layers and bundles we need integration tests. So we implement a simple integration test to show the

concepts. Additionally, it should also be integrated in the Maven build process.

For an easier configuration and implementation, especially the setup and tear down, we use the Spring DM testing framework. They allow us to simplify these steps. For further information about it, read the Spring DM reference guide *[SpringDM]*.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.integrationtest” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.integrationtest”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. We created our integration test project as a normal Java project with Maven support. Through the use of the Spring DM test utilities, we don’t need to convert it to a Plug-in Project. This will be done on-the-fly during the test execution.
8. Now we are going to implement our integration test. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
9. Override its content with the following snippet. We link it also with our parent project to benefit from the common configurations. We add the dependency to the bundles we want to test and bundles we which provides testing classes. Additionally we add Equinox bundles which represent the target platform. They are used by the Spring DM test utilities to run the integration test.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial</groupId>
    <artifactId>tutorial-osgi</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>tutorial-osgi-integrationtest</artifactId>
  <name>OSGi Tutorial - Integration Test</name>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial </groupId>
      <artifactId>tutorial-osgi-service</artifactId>
      <version>${project.version}</version>
```

```
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.herasaf.tutorial</groupId>
      <artifactId>tutorial-osgi-service-impl</artifactId>
      <version>${project.version}</version>
      <scope>test</scope>
    </dependency>

    <!-- Testing dependencies -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>2.5.5</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.aopalliance</groupId>
      <artifactId>com.springsource.org.aopalliance</artifactId>
      <version>1.0.0</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>com.springsource.slf4j.api</artifactId>
      <version>1.5.0</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>com.springsource.slf4j.log4j</artifactId>
      <version>1.5.0</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>
        com.springsource.slf4j.org.apache.commons.logging
      </artifactId>
      <version>1.5.0</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>log4j.osgi</artifactId>
      <version>1.2.15-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>

    <!-- Equinox dependencies -->
    ...

  </dependencies>
</project>
```

Listing 246: The pom.xml of the org.herasaf.tutorial.integrationtest project

10. Additionally we have to add a link from the parent to our integration test

project. Open for that purpose the *pom.xml* file from our parent project.

11. Add the following line between the <modules> tags.
    `<module>../org.herasaf.tutorial.integrationtest</module>`
12. Now we can close the *pom.xml* of the parent project and return to the integration test project.
13. Create now a new source folder with the name *src/test/java*.
14. In this folder, create a new class `org.herasaf.tutorial.integrationtest. TranslationServiceTest`.
15. Open the class and add the following snippet. We have to override the `getTestBundlesNames` method to configure the Spring DM, which additional bundles should be installed in the OSGi framework for the testing purposes.

```
public class TranslationServiceTest extends
    AbstractConfigurableBundleCreatorTests {

  @Override
  protected String[] getTestBundlesNames() {
    return new String[] {
        "org.herasaf.tutorial, tutorial-osgi-service, 1.0.0",
        "org.herasaf.tutorial, tutorial-osgi-service-impl,
            1.0.0" };
  }

  public void testTranslationService() throws Exception {
    ServiceReference reference = bundleContext
        .getServiceReference(TranslationService.class.getName());
    assertNotNull(reference);

    TranslationService service =
        (TranslationService) bundleContext.getService(reference);
    assertNotNull(service);

    String translation = service.getTranslation("appName");
    System.out.println("Translation: " + translation);
    assertEquals("OSGi Tutorial Service", translation);
  }
}
```

Listing 247: The implementation of the TranslationServiceTest

16. Now our integration test is complete. You can run it now in Eclipse as a normal JUnit test or you can try to build the *org.herasaf.tutorial* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

17 Create an RCP application

Overview

In this tutorial we show you on the basis of simple calculator how to create a RCP application step by step. This means we build the application bundle by bundle with the ulterior motive to focus each bundle on its core concerns. The following figure shows the architecture of the application layer by layer.

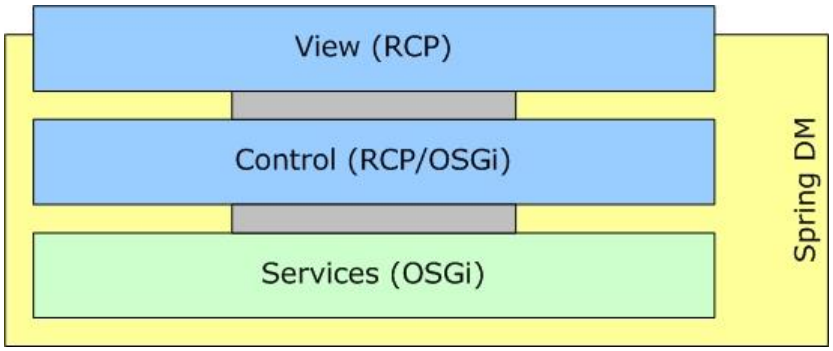

Figure 49: Architecture of the RCP tutorial application

We implement the calculator application in the following steps:

- Creating a parent project
- Creating an OSGi service
- Creating a control layer as the direct OSGi client
- Creating a RCP plug-in
- Creating a Headless PDE Build
- Creating a Feature

For an easier build and development, we use the Maven build tool, which allows the generation of manifest files and loads the required bundles respectively libraries out of the configured repositories We extend it additionally with the Headless PDE Build to generate an executable product.

For the simplification of the development, we use the Spring Dynamic Modules, which supports the declarative import and export of services for OSGi bundles. Without these bundles, we would have to implement the service registration and allocation on our own. To close the gap between RCP and Spring DM, we additionally the Springframework Utilities provided by Martin Lippert *[Lippert08]*.

Conditions

A requirement for a successful implementation of this tutorial is the existence of an infrastructure, how it is described in chapter 8.1, expect for the Automatic Build out of subchapter 8.1.5 we integrate in this tutorial. It's up to you to decide if you want to use the same infrastructure. You can also use substitutes for some tools or different versions, but we cannot guarantee the success.

Additionally we suppose you have some experiences with RCP, OSGi, Spring DM and Maven. Otherwise it is hard to understand the following tutorial.

Notice

Please notice, we don't develop any unit- and integration tests in this tutorial, because we already showed how to implement them in the tutorial described in chapter 16.

Note we give in this tutorial only a small amount of descriptions and explanations about the concepts and decisions. They are all described in detail in the chapters 6and 8.

17.1 Create a parent project

Intention

In a first step we create a parent project. The main purpose of this project is to configure dependencies and options in Maven which should be applied for all of our following projects respectively bundles. It allows also an easy build of all of our projects we will create now.

Tutorial

1. After you opened your Eclipse choose in the menu *File* > *New* > *Other…*
2. In the dialog which opens, select the item *Maven 2 Project* > *Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name "org.herasaf.tutorial.calculator" in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values "org.herasaf.tutorial.calculator".
6. The *Version* field should contain "1.0-SNAPSHOT". Change it to "1.0.0" and click the *Finish* button.
7. Now we created our parent project. Since this project contains only Maven configurations, you can delete the *src* and *target* folder.
8. After this step you should only have the *Maven Classpath Container*, the *JRE System Library* and the *pom.xml* file in our project.
9. Open now the *pom.xml* file and change to the source view.
10. Override its content with the following snippet. We define default properties which are used for the maven bundle plug-in, add the Spring DM bundles as shared dependencies, configure the used plug-ins especially the maven bundle plug-in, and add repositories to get the Spring DM bundles.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.herasaf.tutorial.calculator</groupId>
  <artifactId>parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0.0</version>
  <name>Calculator (Parent)</name>
```

```
<properties>
  <manifest.symbolicname>
    ${project.groupId}.${project.artifactId}
  </manifest.symbolicname>

  <manifest.private></manifest.private>
  <manifest.export></manifest.export>
  <manifest.import>
    !${service.export},
    *;resolution:=optional
  </manifest.import>
  <manifest.activator></manifest.activator>
  <manifest.require></manifest.require>
  <manifest.resource></manifest.resource>
  <manifest.classpath>.</manifest.classpath>
  <manifest.environment>J2SE-1.5</manifest.environment>
  <manifest.activationPolicy></manifest.activationPolicy>

  <!-- Other properties -->
  ...
  ...
  <spring.osgi.version>1.1.3</spring.osgi.version>
  <org.eclipse.springframework.util.version>
    1.0.3
  </org.eclipse.springframework.util.version>
</properties>

<modules>
</modules>

<!-- Shared dependencies -->
<dependencies>

  <dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-annotation</artifactId>
    <version>${spring.osgi.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-core</artifactId>
    <version>${spring.osgi.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-extender</artifactId>
    <version>${spring.osgi.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-io</artifactId>
    <version>${spring.osgi.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-mock</artifactId>
    <version>${spring.osgi.version}</version>
    <scope>test</scope>
  </dependency>
```

```
    <dependency>
      <groupId>org.springframework.osgi</groupId>
      <artifactId>spring-osgi-test</artifactId>
      <version>${spring.osgi.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.eclipse.springframework</groupId>
      <artifactId>util</artifactId>
      <version>
        ${org.eclipse.springframework.util.version}
      </version>
    </dependency>

  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-javadoc-plugin</artifactId>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <aggregate>false</aggregate>
          <quiet>true</quiet>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.0.0</version>
        <extensions>true</extensions>
        <configuration>
          <manifestLocation>META-INF</manifestLocation>
          <instructions>
           <Bundle-Name>
             ${artifactId}</Bundle-Name>
```

```
            <Bundle-SymbolicName>
              ${manifest.symbolicname}
            </Bundle-SymbolicName>
            <Import-Package>
              ${manifest.import}
            </Import-Package>
            <Export-Package>
              ${manifest.export}
            </Export-Package>
            <Private-Package>
              ${manifest.private}
            </Private-Package>
            <Bundle-Activator>
              ${manifest.activator}
            </Bundle-Activator>
            <Require-Bundle>
              ${manifest.require}
            </Require-Bundle>
            <Include-Resource>
              ${manifest.resource}
            </Include-Resource>
            <Bundle-RequiredExecutionEnvironment>
              ${manifest.environment}
            </Bundle-RequiredExecutionEnvironment>
            <Bundle-ActivationPolicy>
              ${manifest.activationPolicy}
            </Bundle-ActivationPolicy>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
      <id>com.springsource.repository.bundles.external</id>
      <name>SpringSource Enterprise Bundle Repository - External
Bundle Releases</name>
      <url>
        http://repository.springsource.com/maven/bundles/
        external
      </url>
    </repository>

    <repository>
      <id>com.springsource.repository.bundles.release</id>
      <name>SpringSource Enterprise Bundle Repository - Spring-
Source Bundle Releases</name>
      <url>
        http://repository.springsource.com/maven/bundles/
        release
      </url>
    </repository>

    <repository>
      <id>spring-release</id>
      <name>Spring Portfolio Release Repository</name>
      <url>http://maven.springframework.org/release</url>
    </repository>

    <repository>
      <id>spring-external</id>
      <name>Spring Portfolio External Repository</name>
      <url>http://maven.springframework.org/external</url>
    </repository>
```

```
    <repository>
      <id>spring-milestone</id>
      <name>Spring Portfolio Milestone Repository</name>
      <url>http://maven.springframework.org/milestone</url>
    </repository>

    <repository>
      <id>spring-ext</id>
      <name>Spring External Dependencies Repository</name>
      <url>
        http://springframework.svn.sourceforge.net/svnroot/
        springframework/repos/repo-ext/
      </url>
    </repository>

    <!-- used when building against Spring snapshots -->
    <repository>
      <id>spring-snapshot</id>
      <name>Spring Portfolio Milestone Repository</name>
      <url>http://maven.springframework.org/snapshot</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>

    <repository>
      <id>i21-s3-osgi-repo</id>
      <name>i21 osgi artifacts repo</name>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <url>http://maven.springframework.org/osgi</url>
    </repository>

  </repositories>

</project>
```

Listing 248: The pom.xml of the org.herasaf.tutorial.calculator project

11. After the configuration of Maven in our parent project, we have nothing more to do in this project.

17.2 Create an OSGi service

Intention

Now we want to implement our OSGi service which computes the result for a given calculation string (i.e. “12+6-4”) it is handed over.

We create in this step two bundles, one for the service declaration and one with the concrete implementation of it. The idea behind this design is to separate the declarations from the implementations to provide additional service implementations in a later step or by other developers, which can exchanged easily. If you have the interfaces and the concrete implementations of your services in the same bundle this could be achieve hardly or only with special configurations.

In this tutorial step we only develop one concrete implementation. But feel free to develop additional bundles with other implementations.

Tutorial: Service declaration

1. We begin with the development of the service declaration bundle. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.calculator.services” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.calculator.services”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. Now we created our service declaration project. It’s a normal Java project with Maven support, but we need a Plug-in Project. So we have to convert it. Select the project in the *Project Explorer* and do a right click on it. Choose *PDE Tools > Convert Projects to Plug-in Projects…*
8. In the dialog which opens, select now our project you want to convert and click *Finish*.
9. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don’t use it.
10. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
11. Override its content with the following snippet. We link it with our parent project to benefit from the common configurations. Additionally, we override the property `<manifest.export>`, because we want to export the interface of our service.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial.calculator</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>services</artifactId>
  <name>Calculator - Services</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.export>
      org.herasaf.tutorial.calculator.services
        ;version=${project.version}
    </manifest.export>
  </properties>
</project>
```

Listing 249: The pom.xml of the org.herasaf.tutorial.calculator.services project

12. Additionally we have to add a link from the parent to our service declaration project. Open for that purpose the *pom.xml* file from our parent project.
13. Add the following line between the <modules> tags.
```
<module>
  ../org.herasaf.tutorial.calculator.services
</module>
```
14. Now we can close the *pom.xml* of the parent project and return to the service declaration project.
15. Create in the folder *src/main/java* a new interface `org.herasaf.tutorial.calculator.services.CalculatorService`.
16. Open the interface and add the following method declaration.
```
public double calculate(String expression);
```
17. That's it, we created our service declaration. Try to build the *org.herasaf.tutorial.calculator* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

Tutorial: Service implementation

18. After the creation of the service declaration we can develop the service implementation bundle. Choose in the Eclipse menu *File* > *New* > *Other…*
19. In the dialog which opens, select the item *Maven 2 Project* > *Maven 2 Project Creation Wizard* and click the *Next* button.
20. Now enter the name "org.herasaf.tutorial.calculator.service.impl" in the field *project name*.
21. Select the option *Create a simple java project* and click the *Next* button.
22. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values "org.herasaf.tutorial.calculator.service.impl".
23. The *Version* field should contain "1.0-SNAPSHOT". Change it to "1.0.0" and click the *Finish* button.
24. Now we created our service implementation project. We also need to convert it to a Plug-in Project. Select the project in the *Project Explorer* and do a right click on it. Choose *PDE Tools* > *Convert Projects to Plug-in Projects…*
25. In the dialog which opens, select now our project you want to convert and click *Finish*.
26. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don't use it.
27. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
28. Override its content with the following snippet. We link it also with our parent project to benefit from the common configurations. Additionally, we override properties to generate a correct manifest file. Additionally we add the dependency to the service declaration project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial.calculator</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>service.impl</artifactId>
  <name>Calculator - Service Implementation</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.private>
      org.herasaf.tutorial.calculator*impl*
    </manifest.private>

    <manifest.import>
      org.herasaf.tutorial.calculator.services
        ;version=${project.version}
    </manifest.import>

    <manifest.resource>
      META-INF/spring=META-INF/spring
    </manifest.resource>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial.calculator</groupId>
      <artifactId> services</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```

Listing 250: The pom.xml of the org.herasaf.tutorial.calculator.service.impl project

29. Additionally we have to add a link from the parent to our service implementation project. Open for that purpose the *pom.xml* file from our parent project.
30. Add the following line between the <modules> tags.
    ```
    <module>
      ../org.herasaf.tutorial.calculator.service.impl
    </module>
    ```
31. Now we can close the *pom.xml* of the parent project and return to the service implementation project.
32. Before we can develop our service implementation, we have to run a Maven build on this project to generate the manifest file. Otherwise we cannot access the `CalculatorService` interface.
    Open the console and run the build with the command `mvn org.apache.felix:maven-bundle-plugin:manifest`.
33. When you now check the manifest file you can see the values of the properties we configured in the *pom.xml*. You maybe have to refresh your Project Explorer instead.

34. Now, we implement our service. Create in the folder *src/main/java* a new class `org.herasaf.tutorial.calculator.service.impl.CalculatorServiceImpl`.
35. Open the class and implement the `calculate` method with a specific algorithm to parse the passed expression and to calculate the result.

```
public class CalculatorServiceImpl implements CalculatorService {

  public CalculatorServiceImpl() {

  }

  public double calculate(String expression) {
    ...
    ...
  }
}
```

Listing 251: Snippet of the CalculatorServiceImpl implementation

36. We've implemented now the calculator service but we have to register it in the OSGi service registry. We use the Spring Dynamic Modules to do this.
37. Create a new folder *spring* under the existing folder *META-INF*.
38. Create the file *bundleContext.xml* in the new folder and add the following snippet. We use this file for the normal Spring instantiation and dependency injection.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="calculatorServiceImpl"
    class="org.herasaf.tutorial.calculator.service.impl.
      CalculatorServiceImpl" />

</beans>
```

Listing 252: The calculator service instantiation in the bundleContext.xml

39. Now add an additional file *bundleContext-osgi.xml* in the same folder and add the following snippet. We use this file to register the instantiated service.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <osgi:service id="calculatorService"
    ref="calculatorServiceImpl" interface="org.herasaf.tutorial.
```

```
        calculator.services.CalculatorService" />

</beans>
```

Listing 253: The calculator service registering in the bundleContext-osgi.xml

40. That's it, we created our service implementation. Try to build the *org.herasaf.tutorial.calculator* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

Unit Tests

During the development of your calculator service, you can also implement unit tests for its verification. We don't treat here how you implement them, because such an implementation is already described in chapter 16.4.

17.3 Create a control layer as the OSGi client

Intention

Up to now we implemented the service to calculate an expression string. Now we are going to implement the control layer as the direct client of our service. It will consist of controller and model classes, suggested by the MVC Pattern.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name "org.herasaf.tutorial.calculator.control" in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values "org.herasaf.tutorial.calculator.control".
6. The *Version* field should contain "1.0-SNAPSHOT". Change it to "1.0.0" and click the *Finish* button.
7. We created our client project as a normal Java project with Maven support. So we also have to convert it to *Project Explorer*. Do a right click on the project and choose *PDE Tools > Convert Projects to Plug-in Projects…*
8. In the dialog which opens, select now our project you want to convert and click *Finish*.
9. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don't use it.
10. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
11. Override its content with the following snippet. We link it with our parent project to benefit from the common configurations. Additionally, we over-

ride the properties to generate a correct manifest file and add the dependency to the service declaration project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial.calculator</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>control</artifactId>
  <name>Calculator - Control</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.import>
      org.herasaf.tutorial.calculator.services
        ;version=${project.version},
    </manifest.import>

    <manifest.export>
      org.herasaf.tutorial.calculator.control.controller
        ;version=${project.version},
      org.herasaf.tutorial.calculator.control.model
        ;version=${project.version}
    </manifest.export>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.herasaf.tutorial.calculator</groupId>
      <artifactId>services</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>

</project>
```

Listing 254: The pom.xml of the org.herasaf.tutorial.calculator.control project

12. Additionally we have to add a link from the parent to our control project. Open for that purpose the *pom.xml* file from our parent project.
13. Add the following line between the <modules> tags.
    ```
    <module>
      ../org.herasaf.tutorial.calculator.control
    </module>
    ```
14. Now we can close the *pom.xml* of the parent project and return to the client project.
15. Before we can develop our client using the service declaration, we have to run a Maven build on this project to generate the manifest file. Otherwise we cannot access the `CalculatorService` interface.
    Open the console and run the build with the command `mvn org.apache.felix:maven-bundle-plugin:manifest`.
16. When you now check the manifest file you can see the values of the properties we configured in the *pom.xml*. You maybe have to refresh

your Project Explorer instead.

17. Create now a new class `org.herasaf.tutorial.calculator.control.model.CalculatorModel` in the folder *src/main/java*.
18. Open the class and add the following snippet. We implement a conventional observer for the notification of changes. Important in this class is on the one hand the class attribute `service` and its setter method. They will be set by the dependency injection provided by Spring, used in a later step.

```
public class CalculatorModel extends Observable {

  private double result = 0;
  private CalculatorService service;

  public void setService(CalculatorService service) {
    this.service = service;
  }

  public double getValue() {
    return result;
  }

  public void informObservers() {
    setChanged();
    notifyObservers();
  }

  public void informObservers(Object arg) {
    setChanged();
    notifyObservers(arg);
  }

  public void setValue(double value) {
    this.result = value;

    informObservers(value);
  }

  public void enterExpression(String expression) {
    informObservers(expression);
  }

  public void calculateResult(String expression) {
    // validation of "expression" already assured in service
    double res = service.calculate(expression);
    this.setValue(res);
  }
}
```

Listing 255: The implementation of the CalculatorModel

19. Create additionally a new class `org.herasaf.tutorial.calculator.control.controller.CalculatorController` in the folder *src/main/java*.
20. Open the class and add the following snippet. Like in the model above, important in this class is on the one hand the class attribute `model` and its setter method. They will be set by the dependency injection provided by Spring, used in a later step.

```
public class CalculatorController {

  private CalculatorModel model;
  private String expression = "";

  public CalculatorController() {

  }

  public void calculateResult() {
    if (expression != null && expression.length() > 0) {
        model.calculateResult(expression);
      expression = "";
    }
  }

  public void enterDigit(String digit) {
    if (digit != null && digit.length() > 0) {
      expression += digit;
      model.enterExpression(expression);
    }
  }

  public void registerObserver(Observer o) {
    model.addObserver(o);
  }

  public void setModel(CalculatorModel model) {
    this.model = model;
  }
}
```

Listing 256: The implementation of the CalculatorController

21. After we've implemented the direct service client in the control layer, we can finish the development of the control layer. Try now to build the *org.herasaf.tutorial.calculator* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

Unit Tests

Like during the service development, you can also implement unit tests for this bundle, if you want to verify it. But it's up to you whether you implement some or not.

Integration Tests

Additionally with the implementation of the direct service client with this layer which also builds the last layer before the UI, you can also implement integration tests against it. Like the unit tests, we don't treat this issue here, because they are already described in chapter 16.5.

17.4 Create a RCP plug-in

Intention

The components we implemented up to now are simple OSGi bundles with describes, provides or uses services. Now we try to integrate them with a RCP plug-in to build an application with a full vertical integration. It will consist of the

view classes only, described in the MVC Pattern.

Tutorial: Implementation

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name “org.herasaf.tutorial.calculator.ui” in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values “org.herasaf.tutorial.calculator.ui”.
6. The *Version* field should contain “1.0-SNAPSHOT”. Change it to “1.0.0” and click the *Finish* button.
7. We created our client project as a normal Java project with Maven support. So we also have to convert it to *Project Explorer*. Do a right click on the project and choose *PDE Tools > Convert Projects to Plug-in Projects…*
8. In the dialog which opens, select now our project you want to convert and click *Finish*.
9. Now we have a Plug-in Project. To avoid problems with the class path, remove the *Maven Classpath Container*, we don’t use it.
10. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
11. Override its content with the following snippet. We link it with our parent project to benefit from the common configurations. Additionally, we override the properties to generate a correct manifest file and add the dependency to the used RCP libraries and own bundles.
    Although we used main the *Import-Package* directive, we must add some of the RCP bundles as *Require-Bundle*, because some of the packages are divided into several bundles.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial.calculator</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>ui</artifactId>
  <name>Calculator - UI</name>
  <packaging>bundle</packaging>

  <properties>
    <manifest.symbolicname>
      ${project.groupId}.${project.artifactId};singleton:=true
    </manifest.symbolicname>
```

```
    <manifest.activator>
      org.herasaf.tutorial.calculator.ui.Activator
    </manifest.activator>

    <manifest.import>
      !${manifest.export},
      org.herasaf.tutorial.calculator.control.controller
        ;version="${project.version}",
      org.herasaf.tutorial.calculator.control.model
        ;version="${project.version}",
      org.herasaf.tutorial.calculator.services
        ;version="${project.version}",
      org.eclipse.springframework.util,
      org.eclipse.core.commands.common,
      org.eclipse.core.runtime;version="3.4",
      org.eclipse.equinox.app,
      org.eclipse.jface.action,
      org.eclipse.jface.resource,
      org.eclipse.swt,
      org.eclipse.swt.events,
      org.eclipse.swt.graphics,
      org.eclipse.swt.layout,
      org.eclipse.swt.widgets,
      org.eclipse.ui,
      org.eclipse.ui.actions,
      org.eclipse.ui.application,
      org.eclipse.ui.plugin,
      org.osgi.framework;version="1.4",
      *;resolution:=optional
    </manifest.import>

    <manifest.export>
      org.herasaf.tutorial.calculator.ui
        ;version="${project.version}"
    </manifest.export>

    <manifest.require>
      org.eclipse.ui,
      org.eclipse.ui.workbench,
      org.eclipse.core.runtime;bundle-version="3.4.0",
      org.eclipse.equinox.common,
      org.eclipse.equinox.registry
    </manifest.require>

    <manifest.resource>
      META-INF/spring=META-INF/spring
    </manifest.resource>

    <manifest.activationPolicy>lazy</manifest.activationPolicy>
  </properties>

  <dependencies>
    <!-- application dependencies -->
    <dependency>
      <groupId>org.herasaf.tutorial.calculator</groupId>
      <artifactId>control</artifactId>
      <version>${project.version}</version>
    </dependency>

    <dependency>
      <groupId>org.herasaf.tutorial.calculator</groupId>
      <artifactId>core</artifactId>
      <version>${project.version}</version>
    </dependency>
```

```
<!-- Additional dependencies -->
<dependency>
  <groupId>org.eclipse</groupId>
  <artifactId>jface</artifactId>
  <version>3.4.2.M20090107-0800</version>
</dependency>

<dependency>
  <groupId>org.eclipse</groupId>
  <artifactId>osgi</artifactId>
  <version>3.4.3.R34x_v20081215-1030</version>
</dependency>

<dependency>
  <groupId>org.eclipse.core</groupId>
  <artifactId>commands</artifactId>
  <version>3.4.0.I20080509</version>
</dependency>

<dependency>
  <groupId>org.eclipse.core</groupId>
  <artifactId>runtime</artifactId>
  <version>3.4.0.v20080512</version>
</dependency>

<dependency>
  <groupId>org.eclipse.equinox</groupId>
  <artifactId>app</artifactId>
  <version>1.1.0.v20080421-2006</version>
</dependency>

<dependency>
  <groupId>org.eclipse.equinox</groupId>
  <artifactId>common</artifactId>
  <version>3.4.0.v20080421-2006</version>
</dependency>

<dependency>
  <groupId>org.eclipse.equinox</groupId>
  <artifactId>registry</artifactId>
  <version>3.4.0.v20080516-0950</version>
</dependency>

<dependency>
  <groupId>org.eclipse.ui</groupId>
  <artifactId>workbench</artifactId>
  <version>3.4.2.M20090127-1700</version>
</dependency>

<dependency>
  <groupId>org.eclipse</groupId>
  <artifactId>ui</artifactId>
  <version>3.4.2.M20090204-0800</version>
</dependency>

<dependency>
  <groupId>org.eclipse</groupId>
  <artifactId>swt</artifactId>
  <version>3.4.2.v3452b</version>
</dependency>

<dependency>
  <groupId>org.eclipse.swt.win32.win32</groupId>
  <artifactId>x86</artifactId>
```

```
      <version>3.4.1.v3452b</version>
    </dependency>

    <dependency>
      <groupId>org.eclipse.springframework</groupId>
      <artifactId>util</artifactId>
      <version>1.0.3</version>
    </dependency>
  </dependencies>
</project>
```

Listing 257: The pom.xml of the org.herasaf.tutorial.calculator.ui project

12. Additionally we have to add a link from the parent to our RCP plug-in project. Open for that purpose the *pom.xml* file from our parent project.
13. Add the following line between the <modules> tags.
    ```
    <module>
      ../org.herasaf.tutorial.calculator.ui
    </module>
    ```
14. Now we can close the *pom.xml* of the parent project and return to the client project.
15. Before we can develop our client using the service declaration, we have to run a Maven build on this project to generate the manifest file. Otherwise we cannot access the classes and interface out of the reference bundles. Open the console and run the build with the command `mvn org.apache.felix:maven-bundle-plugin:manifest`.
16. When you now check the manifest file you can see the values of the properties we configured in the *pom.xml*. You maybe have to refresh your Project Explorer instead.
17. Normally you will see now an error in Eclipse, because we configured an activator class in this plug-in, but it doesn't exist yet. To correct that, create now a new class `org.herasaf.tutorial.calculator.ui.Activator` in the folder *src/main/java*.
18. Open the class and add the following snippet.

```
public class Activator extends AbstractUIPlugin {

  // The plug-in ID
  public static final String PLUGIN_ID =
"org.herasaf.calculator.ui";

  // The shared instance
  private static Activator plugin;

  /**
   * The constructor
   */
  public Activator() {
  }

  public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
  }

  public void stop(BundleContext context) throws Exception {
    plugin = null;
```

```
    super.stop(context);
  }

  /**
   * Returns the shared instance
   *
   * @return the shared instance
   */
  public static Activator getDefault() {
    return plugin;
  }
}
```

Listing 258: Snippet of the Activator implementation in the RCP plug-in

19. Now the error in Eclipse should be disappeared. Now we want to implement a view with a numpad and a text field to use the calculator. To do this, we use the extension points. Therefore, open the *Plug-in Manifest Editor* by double-clicking on the Manifest file and change to the tab *Extensions*.
20. At first we must defined the plug-in as an application to allow the implementation of an own application. Click on the *Add…* button.
21. Select the entry *org.eclispe.core.runtime.applications* in the list and click *Finish*.
22. Select the sub element of the entry. The values shown in the *Extension Element Details* must not be changed.
23. Do now a right-click on *(application)* and select *New* > *run* to create a new application implementation for the extension point, like shown in **Figure 50**.

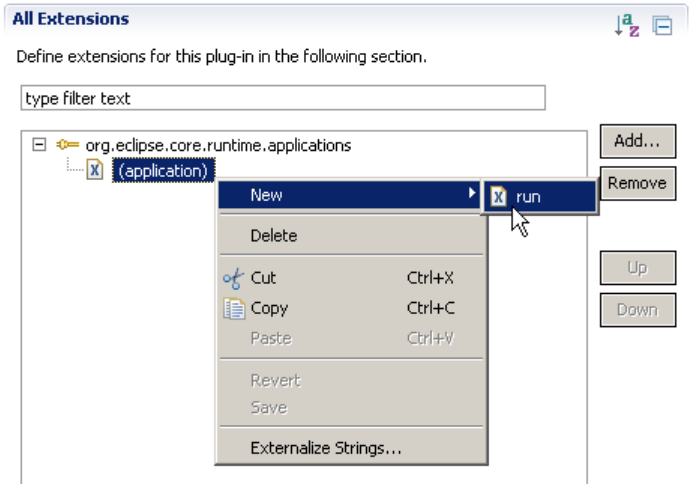

Figure 50: Edit of the extension point org.eclipse.core.runtime.applications

24. Select the new subentry and click in the *Extension Element Details* on the *class** link and create a new class `org.herasaf.tutorial.calculator.ui.Application`. **Figure 51** shows the Extension Element Details view with the configured new application class.

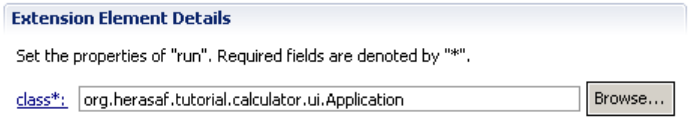

Figure 51: Configuration of the new application class for the extension point

25. Eclipse changes directly to the new created class. We go back to the *Plug-in Manifest Editor*.
26. Click now again on the *Add…* button, because we need to create a perspective, which will contain afterwards our calculator view with the numpad.
27. Select the entry *org.eclispe.ui.perspectives* in the list and click *Finish*.
28. Select now the sub element of the entry *org.eclipse.ui.perspectives*. Change the values in the *Extension Element Details* until they correspond with those shown in **Figure 52**. Therefore you have to create a class `org.herasaf.tutorial.calculator.ui.Perspective`.

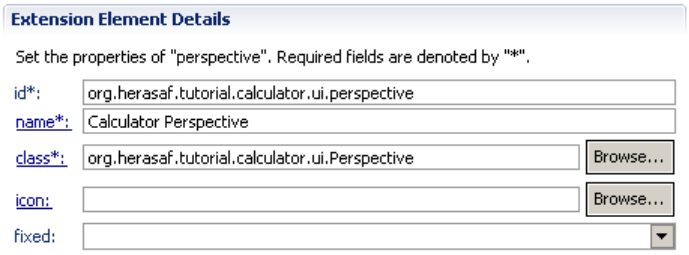

Figure 52: Configuration of the perspective extension point

29. Eclipse changes here also directly to the new created class. We go back to the *Plug-in Manifest Editor*.
30. Click now a last time on the *Add…* button to create and configure our calculator view.
31. Select this time the entry *org.eclispe.ui.views* in the list and click *Finish*.
32. Do a right-click on the new entry and choose *New* > *view* to create our concrete view. **Figure 53** shows how to do that.

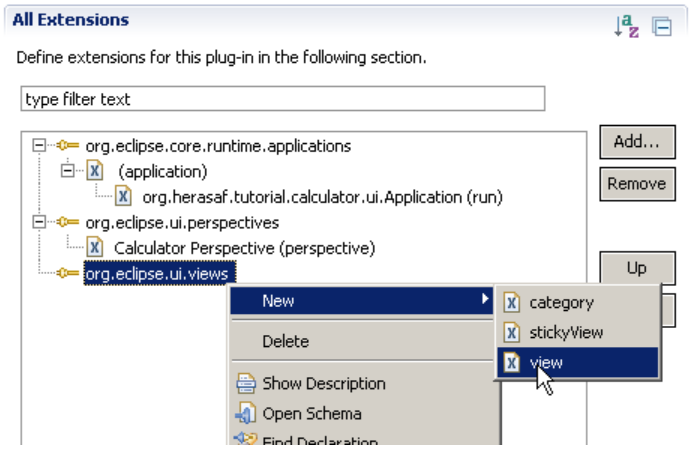

Figure 53: Edit of the extension point org.eclipse.ui.views

33. Select now the sub element of the entry *org.eclipse.ui.views*. Change the values in the *Extension Element Details* until they correspond with those shown in **Figure 54**. Therefore you have to create also a new class `org.herasaf.tutorial.calculator.ui.view.CalculatorView`.

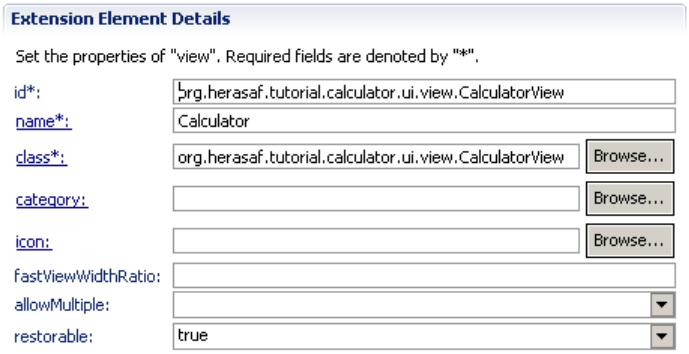

Figure 54: Configuration of the view extension point

34. Now we have to implement the new created classes with new logic and content. Therefore open the class Application and override the methods start and stop with the instructions shown in **Listing 259**.

```
public Object start(IApplicationContext context)
  throws Exception {

  Display display = PlatformUI.createDisplay();
  try {
    int returnCode = PlatformUI.createAndRunWorkbench(display,
          new ApplicationWorkbenchAdvisor());
    if (returnCode == PlatformUI.RETURN_RESTART)
      return IApplication.EXIT_RESTART;
    else
      return IApplication.EXIT_OK;
  } finally {
    display.dispose();
  }
}

public void stop() {
  final IWorkbench workbench = PlatformUI.getWorkbench();
  if (workbench == null)
    return;
  final Display display = workbench.getDisplay();
  display.syncExec(new Runnable() {
    public void run() {
      if (!display.isDisposed())
        workbench.close();
      }
  });
}
```

Listing 259: Snippet of the `Application` class in the RCP plug-in

35. You will get now an error, because the class `ApplicationWorkbenchAdvisor` doesn't exist yet. So we create it in the same package, because it we need a representation of a workbench for our application.
36. Fill the new class with the code snippets shown in **Listing 260**. We configure with this class a window advisor and the initial perspective for our application.

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor
{
  private static final String PERSPECTIVE_ID =
        "org.herasaf.tutorial.calculator.ui.perspective";

  @Override
  public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
    IWorkbenchWindowConfigurer configurer) {
    return new ApplicationWorkbenchWindowAdvisor(configurer);
  }

  @Override
  public String getInitialWindowPerspectiveId() {
    return PERSPECTIVE_ID;
  }
}
```

Listing 260: Snippet of the `ApplicationWorkbenchAdvisor` class in the RCP plug-in

37. You will additionally get an error, because the class `ApplicationWorkbenchWindowConfigurer` doesn't exist yet. We create this class in the same package too, to provide a configuration for the workbench window of our application.
38. Fill the new class with the code snippets shown in **Listing 261**. We can use this class to configure menus, action bars as well as pre- and post-processing of general window events, like *open* or *close*.

```
public class ApplicationWorkbenchWindowAdvisor extends Workben-
chWindowAdvisor {

  public ApplicationWorkbenchWindowAdvisor(
    IWorkbenchWindowConfigurer configurer) {
    super(configurer);
  }

  @Override
  public void preWindowOpen() {
    IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
    configurer.setInitialSize(new Point(400, 300));
    configurer.setShowCoolBar(true);
    configurer.setShowFastViewBars(false);
    configurer.setShowPerspectiveBar(true);
    configurer.setShowProgressIndicator(false);
    configurer.setShowMenuBar(true);
    configurer.setShowStatusLine(false);
    configurer.setTitle("Calculator");
  }
}
```

Listing 261: Snippet of the `ApplicationWorkbenchWindowAdvisor` class in the RCP plug-in

39. Now open the class `Perspective` to implement and configure our perspective.
40. Override its content with the code snippets shown in **Listing 262**. We add with this instructions our view class to the in the editor area of our perspective.

```
public class Perspective implements IPerspectiveFactory {

  public void createInitialLayout(IPageLayout layout) {
    layout.addView(
      "org.herasaf.tutorial.calculator.ui.view.CalculatorView",
      IPageLayout.LEFT,
      IPageLayout.DEFAULT_VIEW_RATIO,
      IPageLayout.ID_EDITOR_AREA);
    layout.setEditorAreaVisible(false);
  }
}
```

Listing 262: Snippet of the `Perspective` class in the RCP plug-in

41. At last, open now the `CalculatorView` class to implement our concrete calculator view. The code snippets in **Listing 263** shows a possible structure of the class.

```
public class CalculatorView extends ViewPart implements Observer {
  private Composite parent = null;

  private Composite digitPanel = null;
  private Text display = null;
  private Button b0 = null;
  private Button b1 = null;
  ...
  private Button plus = null;
  private Button minus = null;
  ...

  // Controller
  private CalculatorController controller;

  public void setController(CalculatorController controller) {
    this.controller = controller;
  }

  public CalculatorView() {
    super();
  }

  @Override
  public void createPartControl(Composite parent) {

    this.parent = parent;

    // Create here your user interface with the single elements
    ...
    controller.registerObserver(this);
  }

  public void setFocus() {
    if (display != null) {
      display.setFocus();
```

```
    }
  }

  public void updateText(final String text) {
    parent.getDisplay().asyncExec(new Runnable() {
      public void run() {
        display.setText(text);
      }
    });
  }

  public void update(Observable o, Object arg) {
    this.updateText(arg.toString());
  }
}
```

Listing 263: Snippet of the `CalculatorView` class in the RCP plug-in

42. With the `CalculatorView` we can finish the implementation o f the RCP plug-in.

Tutorial: Configuration

We've implemented now a conventional RCP plug-in containing a typical plug-in structure. To run it as a RCP application we have to do some configurations additionally to implementation. These steps are covered in the steps below.

43. We start with the Spring and Spring DM configuration. Therefore create a new folder *spring* under the existing folder *META-INF*.
44. Create the file *bundleContext.xml* in the new folder and add the snippet shown in **Listing 264**. We use this file for the normal Spring instantiation and dependency injection.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="calculatorModel"
    class="org.herasaf.tutorial.calculator.control.
          model.CalculatorModel">
    <property name="service" ref="calculatorService" />
  </bean>

  <bean id="calculatorController"
    class="org.herasaf.tutorial.calculator.control.
          controller.CalculatorController">
    <property name="model" ref="calculatorModel" />
  </bean>

  <bean id="calculatorView"
    class="org.herasaf.tutorial.calculator.ui.view.
          CalculatorView"
    scope="prototype">
    <property name="controller" ref="calculatorController" />
  </bean>
</beans>
```

Listing 264: The client instantiation in the bundleContext.xml

45. Now add an additional file *bundleContext-osgi.xml* in the same folder and add the snippet shown in **Listing 265**. We use this file to get the reference of the service.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <osgi:reference id="calculatorService"
    interface="org.herasaf.tutorial.calculator.services.
               CalculatorService" />

</beans>
```

Listing 265: The calculator service reference in the bundleContext-osgi.xml

46. We continue with the definition of the product for its generation. This step allows us to define this bundle as the branding plug-in, or the main plug-in so to speak, for a application wide configuration. For this purpose we have to create a *Product Configuration* in the RCP plug-in.
47. Create a new *Product Configuration* using the Eclipse menu *File* > *New* > *Product Configuration*.
48. In the creation dialog for the *Product Configuration* select our RCP plug-in project in the upper part.
49. In the lower part, enter the name “calculator.product” in the field *File name* and select the option *Create a configuration file with basic settings*. Afterwards click *Finish*.
50. Open the product file, if it isn’t still open. It should show you the *Overivew* tab.
51. Fill now the four fields in the mask with the values shown in **Figure 55**. Note, for entering the value of the field ID, you have to create a new product definition. Use the *New…* button for that. Afterwards a dialog for the creation of a new product definition should open. Fill its fields in the dialog with the data shown in **Figure 56**.
52. Move now to the *Configuration* tab. Here we have to add the plug-ins and fragments which constitute the whole application.
53. Click the button *Add…* button and choose the bundles and plug-ins we implemented in the previous steps.
54. Eclipse provides now the feature to compute the dependencies from the already added bundles and plug-ins and to add it additionally. Click therefore the *Add Required Plug-ins* button. For safety, check if the Spring Framework bundles are also added automatically. If not, add them by hand using the *Add…* button. To resolve the dependencies of Spring, click again the *Add Required Plug-ins* button.
55. Those are the product configurations. The rest of the configurations we can ignore at the moment.

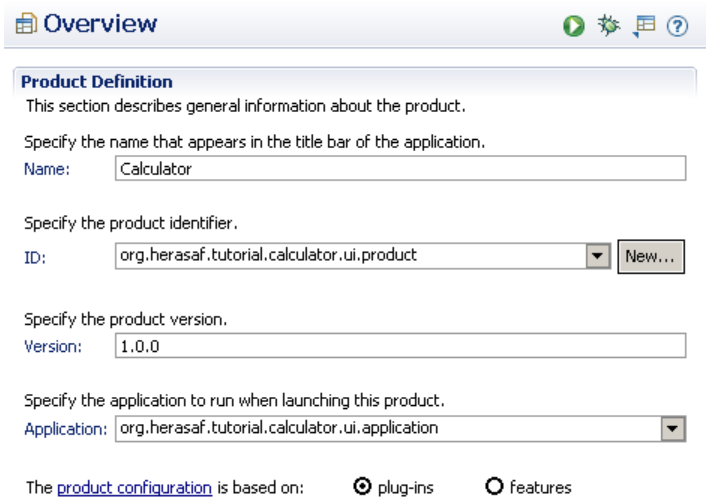

Figure 55: Configuration of the product in the RCP plug-in

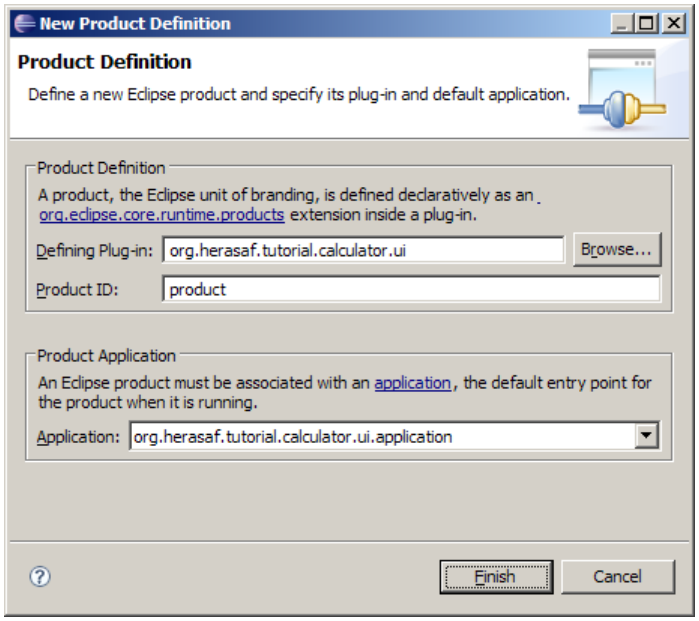

Figure 56: Dialog to create a product identifier

56. Since the RCP application doesn't start all plug-ins and bundles defined in the product automatically, especially those of Spring, we have to do an additional configuration.
    Create a new file *config.ini* in the root directory of your RCP plug-in. It defines amongst others the plug-ins and bundles which should start during the launch of the application.
57. Open the file and add override its content with the one shown in **Listing 266**.

```
#Product Runtime Configuration File
osgi.splashPath=platform:/base/plugins/org.herasaf.tutorial.↵
```

```
calculator.ui
eclipse.application=org.herasaf.tutorial.calculator.ui.application
eclipse.product=org.herasaf.tutorial.calculator.ui.product
osgi.bundles=org.eclipse.equinox.common@2:start,↵
org.eclipse.update.configurator@3:start,↵
org.eclipse.core.runtime@start,↵
org.springframework.bundle.osgi.extender@start,↵
org.springframework.bundle.osgi.io@start,↵
org.springframework.bundle.osgi.core@start,↵
org.herasaf.tutorial.calculator.control@start,↵
org.herasaf.tutorial.calculator.services@start,↵
org.herasaf.tutorial.calculator.service.impl@start
osgi.bundles.defaultStartLevel=4
```

Listing 266: Configuration of the config.ini for the correct application startup

58. We have to link the run startup configuration with the one of the product. Open the *calculator.product* file and move to the *Configuration* tab.
59. In the lower part, you can configure the *Configuration File*. Select the option *Use an existing config.ini file* and choose the *config.ini* file we've previously created.
60. At last, we want to configure the customization of the plug-in, to provide the improved Eclipse look&feel. Create a new file with the name *plugin_customization.ini* in the root directory of the RCP plug-in.
61. Open the file and override its content with the following look&feel settings.
    ```
    org.eclipse.ui/DOCK_PERSPECTIVE_BAR=topRight
    org.eclipse.ui/SHOW_TRADITIONAL_STYLE_TABS=false
    org.eclipse.ui/SHOW_PROGRESS_ON_STARTUP=false
    ```
62. Also this customization has to be linked with the existing configuration. Therefore open the *plugin.xml* file and move to the tab *Extensions*.
63. Foremost you can recognize an additional extension with the name *org.eclipse.core.runtime.products*. This one generated while we created of the product configuration.
64. Select the entry and expand it. Add a new property to the existing *Calculator (product)* through a right-click and *New* > *product*.
65. Select the new property and fill the fields in the *Extension Element Details* with the values shown in **Figure 57**.

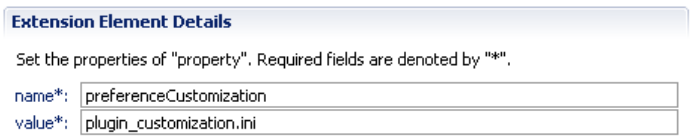

Figure 57: Configuration of the plug-in customization

66. That's it, we implemented and configured our RCP plug-in. Try to build the *org.herasaf.tutorial.calculator* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.
67. Now we can execute our application in Eclipse. Open the file *calculator.product* and move to the *Overview* tab.

68. In the lower part you can recognize the *Testing* part. Click there on the *Synchronize* link.
69. Afterwards you can run the application with a click on the link *Launch an Eclipse application*.
70. Now you should see open the application we implemented.

17.5 Create a Headless PDE Build

Intention

In the previous steps, we implemented and configured an example of a simple RCP application using OSGi managed services. You can run it easily in Eclipse.

In the next step, we're going to implement a Headless PDE Build for the automatic product generation. This covers not only the build of bundles and plug-ins but also the generation of a delivery unit with an executable. We use the PDE Build provided by Eclipse to implement our Headless PDE Build.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Maven 2 Project > Maven 2 Project Creation Wizard* and click the *Next* button.
3. Now enter the name "org.herasaf.pde.pdebuild" in the field *project name*.
4. Select the option *Create a simple java project* and click the *Next* button.
5. Check the values propose in the Maven Project Information. The *Group ID*, *Artifact ID* and *Package name* should contain the values "org.herasaf.pde.pdebuild".
6. The *Version* field should contain "1.0-SNAPSHOT". Change it to "1.0.0" and click the *Finish* button.
7. We created our Headless PDE Build project as a normal Java project with Maven support. We don't need to convert it to a plug-in project.
8. Now we are going to implement our build project. In a first step we want to configure it. So open the *pom.xml* file and change to the source view.
9. Override its content with the following snippet. We simple change its packaging type to *pom*.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.herasaf.pde</groupId>
  <artifactId>pdebuild</artifactId>
  <packaging>pom</packaging>
  <version>1.0.0</version>
  <name>Headless PDE Build</name>
```

```
</project>
```

Listing 267: The pom.xml of the org.herasaf.pde.pdebuild project

10. Additionally we have to add a link from the parent project of your application to our Headless PDE Build to ensure it is available in the Maven build.
11. Add the following line between the <modules> tags.
    ```
    <module>
      ../org.herasaf.pde.pdebuild
    </module>
    ```
12. Now we can close the *pom.xml* of the parent project and return to the build project.
13. Remove now all folders in the project, because we don't use them. Only the *JRE System Library*, maybe the *Maven Classpath Container* and the *pom.xml* should still remain.
14. Now we start with the implementation of the Headless PDE Build. Create a new Ant file *build.xml* in the root directory of the build project, because the PDE Build provided by Eclipse is also implemented as an Ant script.
15. Open the script and override its content with the instructions shown in **Listing 268**. Beside the call of the real PDE Build in the target *pde-build*, we do some pre- and post-processing to allow a more seamless integration.

```
<?xml version="1.0"?>
<!--
    This program and the accompanying materials are made
    available under the terms of the Eclipse Public License v1.0
    which accompanies this distribution, and is available at
      http://www.eclipse.org/legal/epl-v10.html

    This build script creates a build directory containing the
    plugins and features to be built, and then kicks off the PDE
    build process. You could just as easily do this from a shell
    script or cron job.

    Also, the script can be run inside the Eclipse IDE by
    Choosing Run As -> Ant Build from the context menu. It could
    obviously be run outside of the IDE if you have ANT installed
    on your path.
-->
<project default="build">
  <property file="build.properties" />
  <!--
        PDE Build expects that the build directory contains a
        "plugins" directory and a "features" directory. These
        directories should contain the various plug-ins and
        features to be built.

        It's possible to use the CVS checkout process that is
        built into PDE Build. This is done with map files and is
        beyond the scope of this tutorial.

        This tutorial simply copies the projects directly from
        your workspace into the appropriate build directory
```

```
      folders.
  -->
<target name="init">
  <mkdir dir="${buildDirectory}" />
  <mkdir dir="${buildDirectory}/plugins" />
  <mkdir dir="${buildDirectory}/features" />

  <echo message="${projectPrefix}" />

  <copy todir="${buildDirectory}/plugins">
    <fileset dir="../">
      <include name="${projectPrefix}.*/**" />
      <exclude name="${projectPrefix}*feature*/**" />
    </fileset>
  </copy>
  <copy todir="${buildDirectory}/features">
    <fileset dir="../">
      <include name="${projectPrefix}*feature/**" />
    </fileset>
  </copy>
</target>

<!--
      This target actually executes the PDE Build process by
      launching the Eclipse antRunner application.
  -->
<target name="pde-build">
  <echo message="Executing ${eclipseLocation}/plugins/
        org.eclipse.equinox.launcher_
        ${equinoxLauncherPluginVersion}.jar" />
  <java classname="org.eclipse.equinox.launcher.Main"
        fork="true" failonerror="true">
    <arg value="-application" />
    <arg value="org.eclipse.ant.core.antRunner" />
    <arg value="-buildfile" />
    <arg value="${eclipseLocation}/plugins/org.eclipse.
                pde.build_${pdeBuildPluginVersion}
                /scripts/productBuild/productBuild.xml" />
    <arg value="-Dtimestamp=${timestamp}" />
    <arg value="-DallowBinaryCycles=true" />

    <!-- Properties set in the global or project settings -->
    <arg value="-DbuildDirectory=${buildDirectory}" />
    <arg value="-DeclipseLocation=${eclipseLocation}" />
    <arg value="-DequinoxLauncherPluginVersion=
                ${equinoxLauncherPluginVersion}" />
    <arg value="-DpdeBuildPluginVersion=
                ${pdeBuildPluginVersion}" />
    <arg value="-DtopLevelElementType=${topLevelElementType}"/>
    <arg value="-DtopLevelElementId=${topLevelElementId}" />
    <arg value="-Dproduct=${product}" />
    <arg value="-DarchivePrefix=${archivePrefix}" />
    <arg value="-Dconfigs=${configs}" />
    <arg value="-DbuildType=${buildType}" />
    <arg value="-DbuildId=${buildId}" />
    <arg value="-Dbase=${base}" />
    <arg value="-DbaseLocation=${baseLocation}" />
    <arg value="-Dbaseos=${baseos}" />
    <arg value="-Dbasews=${basews}" />
    <arg value="-Dbasearch=${basearch}" />
    <arg value="-DjavacSource=${javacSource}" />
    <arg value="-DjavacTarget=${javacTarget}" />
    <arg value="-DprojectPrefix=${projectPrefix}" />

    <classpath>
```

```
        <pathelement location="${eclipseLocation}/plugins/org.
           eclipse.equinox.launcher_
           ${equinoxLauncherPluginVersion}.jar" />
        </classpath>
    </java>
  </target>

  <!--
        This target actually cleans the build directory after the
        build.
    -->
  <target name="post-build-cleanup">
    <delete includeemptydirs="true">
      <fileset dir="${buildDirectory}">
        <exclude name="${buildLabel}/*.zip" />
      </fileset>
    </delete>
  </target>

  <target name="clean">
    <delete dir="${buildDirectory}" />
  </target>

  <target name="build" depends="init, pde-build, post-build-
cleanup" />
</project>
```

Listing 268: Ant build script for the Headless PDE Build in project org.herasaf.pde.pdebuild

16. Save and close the *build.xml* file.
17. Now we configure the properties to control our Headless PDE Build. In a first step we define properties applying for all projects. Therefore create a new file *build.properties* also in the root directory of your build project.
18. Open the file and override its content with the instructions shown in **Listing 269**.

```
############# PRODUCT/PACKAGING CONTROL #############
runPackager=true

# The location underwhich all of the build output will be col-
lected.
collectingFolder=${archivePrefix}

#Arguments to send to the zip executable
zipargs=

#Arguments to send to the tar executable
tarargs=

############### BUILD NAMING CONTROL #################

# Label for the build.  Used in naming the build output
buildLabel=${buildType}.${buildId}

# Timestamp for the build.  Used in naming the build output
timestamp=007

#Enable / disable the generation of a suffix for the features
that use .qualifier. The generated suffix is computed according
to the content of the feature.
generateFeatureVersionSuffix=true
```

```
############# BASE CONTROL #############
#This property indicates whether you want the set of plug-ins and
features to be considered during the build to be limited to the
ones reachable from the features / plugins being built
filteredDependencyCheck=false

#This property indicates whether the resolution should be done in
development mode (i.e. ignore multiple bundles with singletons)
resolution.devMode=false

skipBase=true
eclipseURL=<url for eclipse download site>
eclipseBuildId=<Id of Eclipse build to get>
eclipseBaseURL=${eclipseURL}/eclipse-platform-${eclipseBuildId}-
win32.zip


############# MAP FILE CONTROL ################
# This section defines CVS tags to use when fetching the map
files from the repository.
# If you want to fetch the map file from repository / location,
change the getMapFiles target in the customTargets.xml

skipMaps=true
mapsRepo=:pserver:anonymous@example.com/path/to/repo
mapsRoot=path/to/maps
mapsCheckoutTag=HEAD

#tagMaps=true
mapsTagTag=v${buildId}


############ REPOSITORY CONTROL ###############
# This section defines properties parameterizing the repositories
where plugins, fragments
# bundles and features are being obtained from.

# The tags to use when fetching elements to build.
# By default thebuilder will use whatever is in the maps.
# This value takes the form of a comma separated list of reposi-
tory identifier (like used in the map files) and the
# overriding value
# For example fetchTag=CVS=HEAD, SVN=v20050101
# fetchTag=HEAD
skipFetch=true


############## JAVA COMPILER OPTIONS ##############
# Specify the output format of the compiler log when eclipse jdt
is used
logExtension=.log

# Whether or not to include debug info in the output jars
javacDebugInfo=false

# Whether or not to fail the build if there are compiler errors
javacFailOnError=true

# Enable or disable verbose mode of the compiler
javacVerbose=true

# Extra arguments for the compiler. These are specific to the ja-
va compiler being used.
```

```
#compilerArg=
```

Listing 269: build.properties for the Headless PDE Build in project org.herasaf.pde.pdebuild

19. Save and close the *build.properties* file.
20. In the next step, we define properties based on your workstation. Therefore, go in your Windows Explorer to the *.m2* directory which contains your *settings.xml* file.
21. Open the file and extend its content with the properties shown in **Listing 270**. Please replace the values of the properties with those applying for your workstation.

```
<settings>
  ...
  <profiles>
    ...
    <profile>
      <id>headlessPDEBuild</id>
      <properties>
        <buildDirectory>
          C:/temp/HeadlessPDEBuild</buildDirectory>
        <eclipseLocation>
          C:/Programme/eclipse
        </eclipseLocation>
        <!-- Version number for org.eclipse.equinox.launcher -->
        <equinoxLauncherPluginVersion>
          1.0.101.R34x_v20081125
        </equinoxLauncherPluginVersion>
        <!-- Version number for org.eclipse.pde.build -->
        <pdeBuildPluginVersion>
          3.4.1.R34x_v20081217
        </pdeBuildPluginVersion>
        <targetPlatformsDirectory>
          ${basedir}/../TargetPlatforms
        </targetPlatformsDirectory>
      </properties>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>headlessPDEBuild</activeProfile>
    ...
  </activeProfiles>
</settings>
```

Listing 270: Workstation-based properties in the settings.xml for the Headless PDE Build

22. That's it; we defined our Headless PDE Build. We have to define only the project-specific properties, but these must be located in a project of the application we build previously.
    Try to build the *org.herasaf.pde.pdebuild* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems.

17.6 Create a Feature

Intention

After the implementation of the project for the Headless PDE Build, we have to integrate it in the build process of our calculator application. We do this through the implementation of a feature project. This has two advantages compared with the integration in the *pom.xml* file of the parent project.

The first is a correct build sequence. If you integrate the Headless PDE Build in the parent project, the PDE Build will be invoked before all other bundles and plug-ins will be built. This sequence is logically wrong. The other one is, you can use the feature project to export your application in Eclipse only as a feature if you don't want a whole application.

Tutorial

1. We begin with the creation of a new project. Choose in the Eclipse menu *File > New > Other…*
2. In the dialog which opens, select the item *Plug-in Development > Feature Project* and click the *Next* button.
3. Now enter the name "org.herasaf.tutorial.calculator.feature" in the field *project name*.
4. Change only the *Feature Name* to "Calculator Feature" and click the *Next* button.
5. Select the option *Initialize from the plug-ins list* and choose the bundles and plug-ins we implemented previously.
6. Click *Finish* to finalize the Feature creation.
7. Open the feature.xml in the Feature Manifest Editor, if it isn't still open.
8. Some of the fields of the Overview tab are filled with data we entered previously. We have to fill the field *Branding Plug-in*. Click on the *Browse…* button and choose our RCP plug-in.
9. Move now to the *Plug-ins* tab. Here you can see the plug-ins and fragments which are contained in the feature.
10. If some of the bundles and plug-ins of our application is missing, add them.
11. Move forward to the *Dependencies* tab. Here we have to define which features and/or plug-ins in the must be present before this feature can be installed.
12. Click the *Compute* button to instruct Eclipse to compute the dependencies automatically.
13. That's it; we don't have to do more configurations. The ones we made are enough.
14. At last, we have to integrate the feature in Maven. Since Eclipse or better our Maven plug-in doesn't provide a conversion from Maven projects to Maven managed feature projects yet, we have to do this by hand.
15. But first, we create define the Maven configuration. Create a new file *pom.xml* in the root directory of the feature project.
16. Override its content with the following snippet. We use the Maven AntRun Plugin to execute the Ant script in our Headless PDE Build project. Before the Ant call we define the properties which should be passed to

the script.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">

  <parent>
    <groupId>org.herasaf.tutorial.calculator</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>feature</artifactId>
  <packaging>pom</packaging>
  <name>Calculator - Feature</name>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <!-- Clean the product build directory -->
          <execution>
            <id>clean-product</id>
            <phase>clean</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <property name="buildDirectory"
                          value="${buildDirectory}" />
                <ant dir="${basedir}/../org.herasaf.pde.pdebuild"
                     target="clean"
                     inheritRefs="true" />
              </tasks>
            </configuration>
          </execution>
          <!-- Execute the product build -->
          <execution>
            <id>build-product</id>
            <phase>install</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <!-- Global settings -->
                <property name="buildDirectory"
                          value="${buildDirectory}" />
                <property name="eclipseLocation"
                          value="${eclipseLocation}" />
                <property name="equinoxLauncherPluginVersion"
                        value="${equinoxLauncherPluginVersion}"/>
                <property name="pdeBuildPluginVersion"
                          value="${pdeBuildPluginVersion}" />
                <!-- Project settings -->
                <property name="topLevelElementType"
                          value="feature" />
                <property name="topLevelElementId"
                          value="Calculator" />
                <property name="product"
```

```
                        value="${basedir}/../org.herasaf.
                              tutorial.calculator.ui/
                              calculator.product" />
                <property name="archivePrefix"
                        value="calculator" />
                <property name="configs"
                        value="win32, win32, x86" />
                <property name="buildType"
                        value="I" />
                <property name="buildId"
                        value="Calculator" />
                <property name="base"
                        value="${targetPlatformsDirectory}" />
                <property name="baseLocation"
                        value="${base}/equinox-3.4.2" />
                <property name="baseos"
                        value="win32" />
                <property name="basews"
                        value="win32" />
                <property name="basearch"
                        value="x86" />
                <property name="javacSource"
                        value="1.5" />
                <property name="javacTarget"
                        value="1.5" />
                <property name="projectPrefix"
                       value="org.herasaf.tutorial.calculator"/>

                <ant dir="${basedir}/../org.herasaf.pde.pdebuild"
                     target="build"
                     inheritRefs="true" />
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 271: The pom.xml of the org.herasaf.tutorial.calculator.feature project

17. Additionally we have to add a link from the parent to our feature project. Open for that purpose the *pom.xml* file from our parent project.
18. Add the following line between the <modules> tags.
    ```
    <module>
      ../org.herasaf.tutorial.calculator.feature
    </module>
    ```
19. Now we can close the *pom.xml* of the parent project and return to the build project.
20. Open the Windows Explorer and navigate to the location of our feature project. Normally this should be the Eclipse workspace.
21. Open the *.project* file and complement it with the snippets shown in **Listing 272**.

```
<projectDescription>
  <name>org.herasaf.tutorial.calculator.feature</name>
  ...
  <buildSpec>
```

```
    ...
    <buildCommand>
      <name>
        org.eclipse.iam.jdt.core.mavenIncrementalBuilder
      </name>
      <arguments>
      </arguments>
    </buildCommand>
  </buildSpec>
  <natures>
    ...
    <nature>
      org.eclipse.iam.jdt.core.mavenNature
    </nature>
  </natures>
</projectDescription>
```

Listing 272: Snippet of the .project file to with the Maven configurations of the project

22. Go now back to your Eclipse and refresh the feature project. It should been Maven managed.
23. That's it; we defined our feature project for the integration of our Headless PDE Build. Try to build the *org.herasaf.tutorial.calculator* project with a Maven build in your console. Use the command `mvn clean install`. If everything is correct, the build should pass without problems and in the *buildDirectory*, an archive with the product should be generated.

Appendix C: Project Management

18 Project Planning

18.1 Introduction

Intention

This chapter contains the project schedule, the risk and quality management, the technologies and standards used in the project as well as a final report.

18.2 Overview

Objectives

This chapter contains the organization of the project. It defines the project roles of each involved person as well as their responsibilities. Additionally, important appointments and further definitions are made.

Organizational structure

Tobias Forster (FOTO)	Responsibilities
Project Manager Programmer	• Project controlling • Communication in team • Design, Tests Email: tobias.forster@herasaf.org

Ylli Sylejmani (YSY)	Responsibilities
Lead programmer Quality manager	• Analysis, Architecture • Refactoring Coordination • Code Quality • Documents Quality Email: ylli.sylejmani@herasaf.org

External contacts

Wolfgang Giersche (GIW)	Contact
Responsible tutor	Email: wolfgang.giersche@herasaf.org

Tutor responsible for the bachelor thesis and contact person for architectural questions and others related to the project.

René Eggenschwiler (EGR)	Contact
Coach	Email: rene.eggenschwiler@herasaf.org

Project coach and contact person for any question or issue related to the project.

Dominik Auf der Maur	Contact
Expert	

Project expert responsible for the review as an external person.

Appointments

These appointments are given by the HSR:

- Project start: 16th February 2009
- Abstract (draft) for examiner: 6th June 2009
- Abstract (final) for Claudia Furrer: 10th June 2009
- Project end: 12th June 2009
- Presentation: 28th August 2009

Meetings

There will be weekly meetings with the project coach (EGR) and if needed also with the responsible tutor (GIW).

These meetings will take place on Friday. The current status and iteration of the project will be reviewed and the next steps will be discussed.

18.3 Schedule

Intention

In the following part, the planned iterations and mile stones are listed which define the project plan used for this bachelor thesis.

18.3.1 Process model

Intention

In this chapter, we define the process model we used for this bachelor thesis. The process model defines the way how to plan and organize the project. It has a direct impact on the planning and execution of the project.

Since this bachelor thesis is a research study, a conventional organization of the project like RUP proposes it, is not appropriate. The reason is the fact that there are neither defined requirements nor an expected product. Requirements changes during the thesis depending on the experiences and results of the executed steps.

That's why we decided to work in a lightweight agile process like Scrum, which allows a flexible organization of the project. We defined the phases of the project at the beginning of the project, but the detail steps continuously. So the dates and scope of the iterations and milestones in the following chapters have shown continuously.

18.3.2 Iterations

Intention

This chapter describes the iterations into which the whole bachelor thesis is

divided and what actions are done in them.

Iteration Planning

Iteration 1	**Start**	**End**
The setup of the used **infrastructure** with all required applications and programs as well as the initialization of the documentation are part of the first iteration. Additionally the start of the project management happened.	16/02/2009	18/02/2009

Iteration 2	**Start**	**End**
Study the **technology research** to gain experiences and knowhow with Eclipse RCP, OSGi and the Spring Dynamic Modules through the implementation of tutorials. Additionally setup of tutorial applications with the studied technologies including the documentation of experiences and guides.	19/02/2009	27/02/2009

Iteration 3	**Start**	**End**
Continuation of the **technology study** for the implementation of complex tutorials. A primary focus is the study of the Eclipse RCP APIs. Additionally buildup of **automatic builds** to provide an integrated build environment for a continuous integration. This includes an automatic product generation.	28/02/2009	03/04/2009

Iteration 4	**Start**	**End**
Detailed **analysis** and research of the **TextEditor** and **GEF/GMF** and further advanced Eclipse RCP APIs for the implementation of a tutorial combining them. This includes also the documentation of made experiences.	04/04/2009	24/04/2009

Iteration 5	**Start**	**End**
Continuation of the tutorial **implementation** using the Eclipse RCP APIs **TextEditor**, **GEF/GMF** and further more. This includes also the documentation of made experiences.	25/04/2009	22/05/2009

Iteration 6	**Start**	**End**
Refactoring and extension of the tutorial applications with further RCP APIs and **advanced topics**.	23/05/2009	29/05/2009

Iteration 7	Start	End
Finalization of the tutorial implementation and the **documentation** of the thesis.	30/05/2009	12/06/2009

18.3.3 Milestones

Intention

This chapter lists the milestones set in project planning of the bachelor thesis. Each milestone describes the date on which it has to be reached and what results are expected.

Milestones

M1: Infrastructure & Technology Basics	Date
The infrastructure is setup. Additionally getting familiar with RCP, OSGi and Spring DM is achieved as well as the first tutorial applications are implemented documented.	27/02/2009

M2: Automatic Build	Date
The automatic build is developed and documented in full detail for the further technology research and the development of tutorial applications.	03/04/2009

M3: TextEditor & GEF/GMF tutorial	Date
A tutorial application using the TextEditor and GEF/GMF is implemented, combining both technologies.	24/04/2009

M4: Running Person Locator application	Date
Completion of the Person Locator application through the combination of the text and the graphical editor for the input of structured text and its conversion into a domain model structure. This covers also the documentation.	22/05/2009

M5: Finish bachelor thesis	Date
The bachelor thesis is finished. The completed documentation with all necessary parts about the analysis, solution and developer's guide is delivered to the tutor.	12/06/2009

18.4 Risk management

Introduction

The following chapter describes the risk management with its risks. It shows consequences, actions to prevent them and estimations of the possible damage as well as the priority.

The risk management is a process, which will be traversed in an iterative way during the whole project. New risks will be recognized and recorded while existing ones will be revaluated.

Risk	Consequence	Actions	Invest in action (working hours)	Maximum damage (working hours)	Probability of event in per-cent	Emphasis of damage (working hours)	Priority (1-low, 2-medium, 3.high)
R01: Bad medical condition of a team member	Delay of project	-	0	25	20%	5	2
R02: The technologies are too complex	Delay of project, increased research effort	Good preparation, studying tutorials and books, communicate with the tutor and coach, weekly meetings	35	60	25%	15	3
R03: Maven is not compatible with RCP/OSGi and Spring DM	Maven cannot be used in combination with RCP/OSGi and Spring DM, loss of time	Research and studying of books and tutorials	20	30	50%	15	3
R04: Loss of SVN server infrastructure	Current status of work will be lost	Create local copies regularly	2	10	5%	0.5	2
R05: Unexpected change of project scope	Additional efforts necessary, changes in the existing planning	Regular progress status to the coach and tutor, weekly meetings	5	50	25%	12.5	3
R06: Insufficient or missing technology descriptions	Additional effort for research and acquisition of technology know-how	-	0	60	33%	20	
R07: Missing compatibility between the different technologies	Additional effort for implementing adapters and other mediator techniques	Extensive research to find such incompatibilities	5	50	25%	12.5	
Total working hours contained in work package			**67.0**				
Total accruals						**80.5**	

18.5 Quality management

Documentation The project will be documented in the following documents:

- 2009_Spring_BachelorThesis_RCPRA_Thesis.doc

The documentation will be updated permanently during all the phases of the project. The language in the documentation is English for all documents.

Tests

For all the modules and features complete tests will be written. Mock objects will be used to test each part individually.

Reviews / Communication

Long range architecture and design decisions will be made by the whole team. In special cases the coach and tutor will take part in the deciding process.

Weekly meetings with the coach and/or tutor will be held to keep the progress transparent and for communication.

To ensure the quality and correctness of the documents, each team member will read all changes in the documents. Additionally a code review will take place every week to ensure the quality of the code.

The chapter *Minutes of meeting* will contain all decisions and meeting topics.

Version Management

All project products including its source code will be managed by a SVN version control system.

Build Automation

Maven2 will be used for project automation and builds.

18.6 Technologies and Standards

18.6.1 Naming conventions

Documents

The documents written in this bachelor thesis will have the following name structure to ensure a uniform naming:

- «year»_«term»_BachelorThesis_RCPRA_«theme».«filetype»

An example for this naming convention is shown below:

- 2009_Spring_BachelorThesis_RCPRA_Thesis.doc

Classes

Classes specially implementing a well known pattern like Adapter, Strategy or so on, should be marked with a corresponding suffix. The below list define some of them:

- Adapter Pattern: «ClassName»Adapter
- Locator Pattern: «ClassName»Locator
- Strategy Pattern: «ClassName»Strategy

18.6.2 Technologies

Equinox x.x

Equinox is the implementation of the OSGi R4 core framework specification developed by the Eclipse Foundation. In version 3.0 it replaced the Eclipse plug-in technology. Through its plug-in system it allows the developers to create

modular applications. *[Equinox]*

Maven 2.x — Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. *[Maven]*

Eclipse RCP x.x — Eclipse RCP is the basic platform of the Eclipse IDE. It consists of Equinox, the core platform, the Standard Widget Toolkit (SWT), JFace and the Eclipse Workbench. Its main purpose is to allow the development of general purpose applications. *[EclipseRCP]*

JUnit x.x — JUnit is one of the first testing frameworks for the Java programming language. Through its simple structure it allows the development of unit and integration tests in straightforward manner. *[JUnit]*

SLF4J x.x — The Simple Logging Façade for Java is a powerful framework providing a Java logging API by means of a simple façade pattern. The underlying logging framework is determined at deployment time. At the moment Log4J, the JDK Logging API and the Jakarta Commons Logging is supported. [SLF4J]

Spring 2.5.x — The Spring Framework (or Spring for short) is an open source application framework for the Java platform.

Although the Spring Framework does not enforce any specific programming model, it has become popular in the Java community as an alternative, replacement, or even addition to the Enterprise JavaBean (EJB) model. *[Spring]*

Spring DM 1.1.x — The Spring Dynamic Modules makes it easy to build Spring applications that run in an OSGi framework, like Equinox. It is an extension of the basic Spring Framework. Beside other utilities, it allows the registration and lookup of OSGi services declaratively in the application context as a main feature. *[SpringDM]*

TestNG 5.x — TestNG is a testing framework for the Java programming language inspired by JUnit and NUnit but introducing some new functionality that purport to make it more powerful and easier to use.

TestNG is designed to cover all categories of tests, including unit, functional, and integration tests. *[wikiTestNG]*

18.6.3 Tools

Eclipse PDE — The Eclipse Plug-in Development Environment provides an appropriate platform with useful tools for the development of Eclipse products. This covers beside simple plug-ins also fragments, features or whole applications. It integrates the development with an execution environment and further features. *[EclipsePDE]*

Maven2 — Maven is a tool for the management and automation of the build of Java applications. It is very similar to the functionality of Apache Ant but is based on different concepts. It uses an integrated dependency management for the resolution of

libraries from a central repository and allows therefore an easier handling of them. *[Maven]*

Ant

Apache Ant is a software tool for the automation of Java software build processes and requires therefore a Java platform. The build process itself is described in a XML file. Accessing version management tools is also possible like the creation of Java archives. *[ApacheAnt]*

18.7 Standards

OSGi

The OSGi core specification *[OSGiCore]* is an open standard that describes a technology for a universal middleware. It provides a service-oriented and component-based environment for development of modular applications in a standardized way. Through the management of the software lifecycle a hot plugging is possible, which allows the installation and deinstallation of modules, called bundles, without a restart of the framework.

The functionality is complemented through the components defined in the OSGi service compendium *[OSGiComp]*. This specification covers a various services for different purposes, such as a logging framework and further more.

18.8 Project management final report

Schedule

Because this bachelor thesis was a research project, the goals at the beginning were very general. Therefore they had to be defined and worked out in detail during thesis why we achieved them successfully.

The team worked more than scheduled during the whole bachelor thesis. The cause was on the one hand the continuous fixation of the goals. On the other hand the occurrence of challenges and problems required a higher research effort. Additionally the effort for the documentation was higher than expected which caused additional efforts.

19 Evaluation

19.1 Introduction

Intention

The evaluation shows the state of the bachelor thesis. It gives a good overview and feedback about the accomplished work of the project team.

The code evaluation was made in a late project state, where the main implementation and redesign tasks were done and the code was frozen. In contrast the time evaluation was made near the completion of the bachelor thesis to reach an exact evaluation result.

19.2 Architecture and code quality

Overview

This chapter contains the evaluation of the source code developed during the bachelor thesis. It is divided into the different tutorial applications. Further splits were not made, because of the limited size of the applications.

19.2.1 OSGi Tutorial

General

This evaluation in this chapter covers the OSGi tutorial application. In exception of the *Number of test code lines*, the values for each metric show only the evaluation results for the productive code, not the test code. The number of code lines does not include the Javadoc.

Metrics

Metrics	
Number of packages	6
Number of interfaces	1
Number of classes	2
Number of methods	8
Number of productive code lines	49
Number of test code lines	70

19.2.2 Basic RCP Application

General

This evaluation in this chapter covers the basic RCP tutorial application. In exception of the *Number of test code lines*, the values for each metric show only the evaluation results for the productive code, not the test code. The number of code lines does not include the Javadoc.

Metrics

Metrics	
Number of packages	15
Number of interfaces	2
Number of classes	20
Number of methods	67
Number of productive code lines	584
Number of test code lines	143

19.2.3 Advanced RCP Application

General

This evaluation in this chapter covers the advanced RCP tutorial application. In exception of the *Number of test code lines*, the values for each metric show only the evaluation results for the productive code, not the test code. The number of

code lines does not include the Javadoc.

Metrics

Metrics	
Number of packages	55
Number of interfaces	20
Number of classes	137
Number of methods	663
Number of productive code lines	5651
Number of test code lines	142

19.3 Time evaluation

General

The duration of this bachelor thesis project was 14 part-time weeks during the whole spring term and additional two fulltime weeks. During the part-time weeks 24 hours per person was estimated and scheduled as well as 40 hours per person in the fulltime weeks.

Overview

	Scheduled	Real
Working hours	720	1175
Average hours per week	45-48	73.5

Registered time per phase

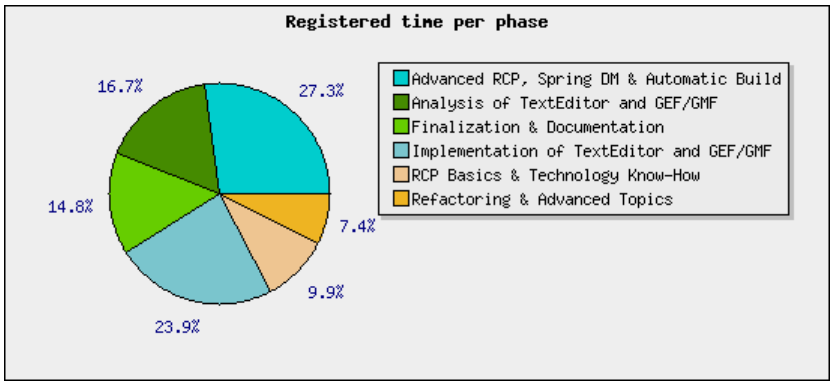

Figure 58: Diagram of the registered time per phase

Registered time per activity

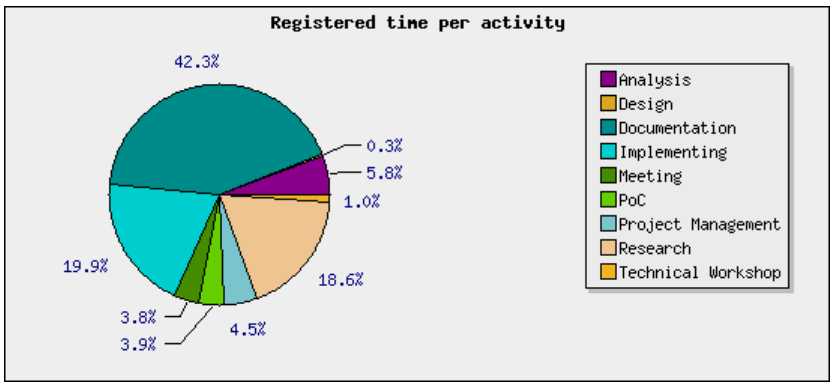

Figure 59: Diagram of the registered time per activity

Appendix D: Minutes of Meetings

20 Meeting Information 14/02/2009

Date: 14/02/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Skype
Start: 03:00 pm
End: 04:15 pm
Keeper of the minutes: Tobias Forster

20.1 Agenda items

Item	Assignment / Information

Administrative issues	• Clarified administrative issues like the weekly effort and the important appointments
	  ○ Start date: 16/02/2009
	  ○ End date: 12/06/2009
	  ○ Duration: 16 weeks
	  ○ Weekly effort: 24 hours per person
	  ○ Total effort: 360 hours per person
	  ○ Other important appointments: http://www.hsr.ch/Diplom-Bachelor-und-Studien.1825.0.html

Aims of the thesis	• Discussed and defined the detailed aims of the Bachelor thesis with its parts and the intention to do this
	  ○ Get familiar with the RCP, OSGi and Spring DM technologies
	  ○ Implementation of a reference architecture
	  ○ Develop concepts for deployment/releasing, layer architecture, modularization and packaging
	  ○ Write a developer's guide with the gained knowledge and best practice in English
	  ○ Write a documentation of the thesis in English

Infrastructure	• Defined the required infrastructure for the Bachelor thesis.
	  ○ Equinox (OSGi implementation)
	  ○ Eclipse RCP
	  ○ Spring Dynamic Modules
	  ○ Maven 2 (including required Eclipse plugins)

Goals for the next two weeks	• Setup the infrastructure of the laboratory computers as well as the notebooks. • Get familiar with RCP and OSGi with its concepts. • Implementation of an example RCP application "Calculator" with the four basic functions to build up a basic knowledge. • Come in touch with the Spring Dynamic Modules technology.

21 Meeting Information 27/02/2009

Date: 27/02/2009
Attendee: Wolfgang Giersche, René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 02:30 pm
End: 04:30 pm
Keeper of the minutes: Tobias Forster

21.1 Agenda items

Item	Assignment / Information
Working progress	• Explained the tasks we worked on in the last two weeks. This includes the setup of the development infrastructure, the research und study of the technologies (RCP, OSGi and Spring DM) and the implementation of example applications. • Presented the example applications and made a detailed walkthrough of them to find parts, which should be refactored and to discuss about the chosen design. • Explained the work on a target platform. The aim of such a definition is a common target platform for each developer using RCP, OSGi and Spring DM. Additionally the platform can easily be updated to new versions. The integration of the Spring DM bundles was not successful until now, that's why we delayed this work to a later point.
Hints	• In the bachelor thesis, where the best practice guide is a very important part, the detailed recording and referencing of all information sources is needed. ○ Additionally all or the important possible solutions should be collected and balanced against each other. This helps the reader to get also different solutions and allows him to understand why a specific solution is recommended. The comparisons of the alternatives should be made in a very detailed way. • If abstract and/or common parts of the applications especially in the UI layer exists, an outsourcing of them in own bundles should be considered. This could lead to a better reusability of these parts. • The UI should be separated as much as possible from the other bundles of the applications. This allows an easier way to emulate the UI with unit test cases. • The UI should not contain any logic, especially any business logic. They should be contained in business services or controllers. Therefore, the UI should only invoke methods on them. • The thesis document should contain the administrative parts of the thesis like the management summary, common explanations, test definitions, reports and so on, whereas the developer's guide contains a guide for the different parts of the system. They contain the recommended solution, a detailed comparison of alternatives and the decision why the recommended solution was chosen.

Points of criticism	• For a proper planning process in JIRA more tasks should be registered and they should be finer grained.
	• The following points of criticism raised during the code walkthrough:
	  ○ It should be considered if the implementation of the undo functionality is placed right in the Activator class.
	  ○ The buttons in the calculator are implemented all in the same way. It should be considered if the definition of a calculator button can reduce the complexity of the code base.

	○ The core bundle should include any RCP libraries. It reduces the coupling if the classes are simple java objects. ○ All bundles which don't contain UI source code should not have an activator class, because it's a RCP class. Additionally OSGi bundles usually don't have any activator class. If such an activation is needed, an own implementation with a start and stop method should be provided which can be used with the *init-method* and *destroy-method* attribute in the bean definition of the Spring configuration files. Another solution is implementing the *BundleActivator* interface which is independent from RCP.
Planning	• Week 1 - 2: RCP Basics & Technology Know-How • Week 3 - 5: Advanced RCP & Spring DM • Week 6 - 7: Analysis of the TextEditor and GEF • Week 8 - 10: Implementation of the TextEditor and GEF view • Week 11 - 12: Refactoring • Week 13 - 16: Advanced TextEditor & Documentation finish
Goals for the next three weeks	• Combine the development of applications with RCP, OSGi and Spring DM additionally with Maven to get a complete development process. For the integration in the OSGi development, the Apache Felix implementation provides Maven plug-ins for this purpose. • The Eclipse Data Binding should be studied and applied in the example applications, because it allows a lower coupling of the UI and control bundles. • The Calculator example application should be refactored, especially same code bases and inner classes should be checked. Additionally the view must be split in three independent classes. (1 model / 3 views) Also the UI and the business logic should be separated more precisely. • Best practices and examples of unit and integration tests should be considered and implemented. • For other developers a read me file in SVN and/or Wiki should be written, which contains the required installation steps for the infrastructure. • The RCP APIs for the Error Log and the Progress/Status should be studied and applied.
Remaining todo's	• The Task API and the Status Manager API should be studied. • The Forms API should be studied. • A best practice for the declaration and implementation of services should be considered. Should the interfaces and implementation be combined in the same bundle or be separated? • The source code editor which is provided by the Eclipse Face should be studied.

22 Meeting Information 06/03/2009

Date: 06/03/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 01:30 pm
End: 04:00 pm
Keeper of the minutes: Tobias Forster

22.1 Agenda items

Item	Assignment / Information
Working progress	• Explained the tasks we were working on last week. This includes the integration of the Apache Felix Bundle Plug-in for Maven in an existing OSGi bundle using Spring DM, the study of Eclipse APIs and test utilities for OSGi. • The integration of the Bundle Plug-in for Maven provides an automatic Manifest generation. But this has to be initiated by a maven command and will not automatically process in case of a pom.xml change. But Eclipse provides a possibility to initiate a maven build when saving or building the project though the definition of an Eclipse Builder in a project. • The test framework which is provided by Spring DM gives a huge simplification of testing OSGi bundles. • We get access to the HERAS^{AF} Wiki where we add tutorials, links and other resources which are worth of sharing. We also get accounts for the HERAS^{AF} repository to add libraries and bundles which are not available in other maven repositories. • The developer's guide was extended with parts and experiences to about RCP, Spring DM and OSGi.
Forum / Wiki	• We decided with René to use mainly the Wiki to offer tutorials of the technologies for others and not the forum. For that reason, a new public space "Tutorials" will be created.
Build and test	• We came to the decision to enlarge the project scope with the setup of an automatic build infrastructure. The aim should be that automatic tests can be process by Bamboo, when source code is checked in. • Additional the test lifecycle should first test the basic bundles and go step by step one layer up until all bundles are tested (bottom up). • The tests should also cover the UI which means the development of UI tests. This can be possible with the Eclipse Test & Performance Tool Platform, or TPTP for short. The plug-in allows recording the interaction with a UI and to proceed it later to test user interface. • Testing of bundles should be made with Spring DM test framework and the correct definition of the pom.xml scopes. This is not the perfect solution, but it is the only one which is possible at the moment, because the Bundle Plug-in for Maven doesn't yet support a comprehensive test lifecycle.
Documentation	• With René we decided to concentrate the different parts of the documentation in the thesis document. We defined a new structure for the documentation which allows the integration of all necessary parts.
Planning changes	• In the weeks from the Thursday 9th to the Wednesday 15th April no lectures will be hold. This week was not considered until now and makes it possible to handle some of the additional tasks. • In the second iteration, before extending and refactoring the Calculator, the tasks for the automatic build should be made.
Changes of the goals	• Instead of GEF, the GMF form Eclipse should be analyzed. It provides

for the next two weeks	a wider range of functionality which can be useful. • The extension of the Calculator tutorial should be kept as minimal as possible. But the main technologies should be integrated in the correct manner. • The Error Log API from Eclipse should be used with the Simple Logging Façade for Java, or SLF4J for short.

23 Meeting Information 13/03/2009

Date: 13/03/2009
Attendee: Wolfgang Giersche, René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 10:30 am
End: 11:30 am
Keeper of the minutes: Ylli Sylejmani

23.1 Agenda items

Item	Assignment / Information
Working progress	• We reported about the progress in the integration- and unit testing. The integration test already works and is partly documented, whereas for the unit testing still a tutorial has to be created. • The definition of the target platform was completed. A maven project includes all the required dependencies for the target platform in its pom.xml file and makes sure the packages are available in the local repository. This is needed to perform integration tests using the Spring DM test framework. • The structure of the documentation has been revised. We presented the new buildup of the documentation. • Different topics have been documented since last week and are included in the thesis documentation of the project. • We are working on the automated build, which is still in progress.
Documentation structure	• The presented new structure of the thesis documentation has been accepted. • The content of the chapter “processes” has been clarified. It should contain and describe software engineering processes and also define a general process model describing how e.g. to integrate a new view in the whole application.
Wiki public space	• A JIRA task has been entered for the creation of a public wiki space where we can publish different topics related to the thesis.
Bamboo workspace	• To move the whole building process on the Bamboo server a Bamboo workspace needs to be created. The workspace will be arranged next week.
Results of an automatic build	• The question “what is the result of a maven build?” was discussed in this meeting. The objective is to be able to create a whole “product” with the automated build. This could be realized by using a self-defined target in maven, such as e.g. “release”. The “export” to a “product” can be realized using an ant script. To execute the “export” a special ant task is provided by Eclipse. More research on this has to be done to achieve the goal during next week.
Goals for next week	• For the next week it’s intended to finish all the tasks related to the automatic build process and to integrate the process into Bamboo. • Another objective for next week is the beginning of the refactoring of the calculator projects.

24 Meeting Information 20/03/2009

Date: 20/03/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 01:30 pm
End: 03:00 pm
Keeper of the minutes: Ylli Sylejmani

24.1 Agenda items

Item	Assignment / Information
Working progress	• The testing tutorial and all the tasks related to it have been finished and documented. • The Wiki public space has been created and is available. • A Bamboo space has been created and the OSGi/Spring DM tutorial has been uploaded including unit- and integration-tests and is running without errors on Bamboo. • We were able to integrate ANT in Maven so that it uses the Eclipse PDE Build tools to build a whole Eclipse RCP product. This means that the product of the build is a ZIP file including a ready to run Eclipse RCP product. • Referring to the refactoring of the calculator project, we started first tasks to source business logic out in own bundles and specify a domain model.
Properties for ANT export script	• The *build.properties* file used by ANT to export an Eclipse RCP product is still containing hard coded paths, versions, etc. A solution has to be found, which makes it possible to pass *pom.xml* properties to it. Project specific properties should be defined in *pom.xml* whereas infrastructure related properties such as the path for the export location of an Eclipse RCP product should be defined in the user specific settings file *settings.xml*.
Goals for next week	• For the next week it's intended to create an example, which merges all existing tutorials until now. This contains the calculator tutorial, including the automatic build of it, the OSGi/Spring DM tutorial, including all unit- and integration tests and the automatic run and build of the whole package. • The example contains an UI layer including the calculator UI as it exists now. A controller layer reads the inputs from the UI, converts the calculations to a string, calls a calculation service and receives results from it. The service provides a method, which executes calculations from a given string and returns the received result. It's intended to contain unit-test as well as an integration test for the service layer. • Next week, the project Expert has been invited for a presentation of our working progress. It's planned to prepare a presentation of the project, the goals achieved until now and to show the example mentioned above. • The tasks related to the automatic build, the implementation of an appropriate application, the documentation and the presentation have received a higher priority. The refactoring of the calculator, the studying of RCP API's and related tasks have been moved to later on according to these circumstances.

25 Meeting Information 27/03/2009

Date: 27/03/2009
Attendee: Wolfgang Giersche, Dominik Auf der Maur, René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Altstetten
Start: 01:00 pm
End: 03:00 pm
Keeper of the minutes: Ylli Sylejmani

25.1 Agenda items

Item	Assignment / Information
Milestone presentation of the working progress	• We presented the actual working progress of our bachelor thesis, showing the logical architecture of the reference architecture, which technologies we used and how we used them, how the build process works and how the dependency tree looks like.
Demonstration of the automatic build	• After the presentation we did a walkthrough through all the projects and showed how they're configured and wired together. We kept the focus on the Spring DM application contexts and the *pom.xml* files. • We demonstrated practically the build process, how it builds all the bundles and packages, runs unit- and integration tests and generates a Zip file including a complete Eclipse executable product.
Feedback from the expert Dominik Auf der Maur	• The expert gave us a positive feedback, saying that the automatic build is well solved, better than other solutions he came across with in practice. • Also, the feedback from the responsible tutor was very positive. He generally was satisfied with our work. What impressed him especially was our solution how we integrated Spring DM in RCP bundles.
Derived tasks from the feedback	• According to the feedback from the participants of this meeting, the following to-do tasks were derived: ○ In the integration tests we use fix names in the method *getTestBundlesNames* to specify the additionally required test bundles. A better solution would be to read these required bundles directly from the *pom.xml* instead of specifying them by hand. ○ In the documentation of the processes and scenarios it's important to think about issues, which could appear in the future. To mention an example, updating the target platform to a newer version, would be such an issue.
Goals for next week	• Finish and improve the automatic build process based on the feedback mentioned above. • Completing the documentation to reach the current state of the working progress. • Finish up and close the first milestone. • Depending on time and workload, start research and studying the different Eclipse RCP API's.

26 Meeting Information 03/04/2009

Date: 03/04/2009
Attendee: Wolfgang Giersche, René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 03:00 pm
End: 06:00 pm
Keeper of the minutes: Ylli Sylejmani

26.1 Agenda items

Item	Assignment / Information
Working progress	• We presented the actual working progress of our bachelor thesis, especially the achievement of milestone *M2: Automatic Build*. • We finalized the Automatic Build with an improved PDE Build. But we couldn't integrate the Headless PDE Build as a JAR like we planned. So it must be available as conventional project furthermore. • Together with Florian Huonder, we set up the Automatic Build on the central Bamboo server including the Headless PDE Build. • Additionally we finalized the documentation of the Automatic Build as well as a first part of the OSGi service layer and a RCP tutorial including the Headless PDE Build. • Also the study and implementation of the Data Binding provided as an API by Eclipse RCP was done. Referring to our own experiences and the discussions with our tutors, its implementation is relatively complex and increases the dependencies between the bundles and classes unnecessarily. Maybe the use of a Java integrated expression language can help simplify this problem.
Presentation of further topics	• Wolfgang Giersche presented its vision of the further steps of our bachelor thesis providing a tutorial application composed of a view with a TextEditor and a view with a GEF/GMF figure. The TextEditor should include a text parser which provides the required Metadata. Additionally the parsed text should be converted into a domain object structure which should be displayed as a GEF/GMF figure.
Planning changes	• We adjusted the planning of the iterations and its core issues like shown in Table 5.

Weeks	Old topics	New topics
1 – 2	RCP Basics & Technology Know-How	RCP Basics & Technology Know-How
~~**3 – 5**~~ **3 – 7**	Advanced RCP & Spring DM	Advanced RCP, Spring DM **& Automatic Build**
~~**6 – 7**~~ **8 – 9**	Analysis of TextEditor and GEF/GMF	Analysis of TextEditor and GEF/GMF *1 week Easter holidays*
~~**8 – 10**~~ **10 – 12**	Implementation of TextEditor and GEF/GMF	Implementation of TextEditor and GEF/GMF
~~**11 – 12**~~ **13 – 14**	Refactoring	Refactoring **& Advanced Topics**
~~**13 – 16**~~ **15 – 17**	Advanced TextEditor & Documentation	**Finalization & Documentation**

Table 5: Planning changes in weekly status meeting (03/04/2009)

• Based on these changes and the presented topics, we decided to move the extensions of the Calculator tutorial application to a later date

	in iteration 5 (*Refactoring & Advanced Topics*) or 6 (*Finalization & Documentation*).
Goals for next two weeks	• Detailed analysis and research of the TextEditor and the GEF/GMF for the implementation of a tutorial corresponding with the topics described above. • Analysis of Wolfgang Giersche's text parser for an own implementation to provide the TextEditor with the required Metadata. • Implementation of a first example implementation using TextEditor and GEF/GMF including further Eclipse RCP APIs and an own text parser.

27 Meeting Information 14/04/2009

Date: 14/04/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Altstetten
Start: 07:00 pm
End: 08:30 pm
Keeper of the minutes: Tobias Forster

27.1 Agenda items

<table>
<tr><th>Item</th><th>Assignment / Information</th></tr>
<tr><td>Working progress</td><td>

We presented the actual working progress of our bachelor thesis with the tasks we started in the last week.
We explained the progress for the research and analysis of the TextEditor and the problems which occurred. Especially the configuration of the TextEditor highlighting and the combination with a domain model was an important issue.
We also explained the progress of the GEF/EMF/GMF research and analysis with the relations of these technologies amongst each other. Also the problems occurred with GMF were presented and discussed.

</td></tr>
<tr><td>Further discussions and decisions</td><td>

We discussed the detailed class structure of the TextEditor with the highlighting and the needed scanners and the integration of external parser. Also the usage of the domain model together with the TextEditor, especially for providing the data for the text highlighting was an important issue.
We also discussed the structure of the GMF diagram editor and its model files configuring and using the basic technologies GEF and EMF. For a better understanding we also presented the proposed class structure of GEF and its configuration in the GMF model files.
Together with René we discussed the generation of the GMF diagram editor and its management, because there are some problems with the generated projects. On the one hand the old projects should be deleted before generating new ones, because we had some trouble with curious class names. On the other hand the projects should be managed by Maven like the other ones. But this is only to achieve manually. Therefore we have to check if we can integrate the generation automatically in the Maven build process and SVN.
A very important discussion was about the domain model, because GMF generates one for itself using EMF. But this one has dependencies to EMF classes. For a simple GMF model this is no problem, but for a general domain model used in other parts of the application such dependencies are not acceptable. So we must research if a linking between the EMF model and the general domain model is possible.

</td></tr>
<tr><td>Planning changes</td><td>

We adjusted the planning of the iterations and its core issues like shown in Table 6. The reason for this change was the higher research and analysis effort for the TextEditor and GEF/GMF as expected.

<table>
<tr><th>Weeks</th><th>Old topics</th><th>New topics</th></tr>
<tr><td>1 – 2</td><td>RCP Basics & Technology Know-How</td><td>RCP Basics & Technology Know-How</td></tr>
<tr><td>3 – 7</td><td>Advanced RCP, Spring DM & Automatic Build</td><td>Advanced RCP, Spring DM & Automatic Build</td></tr>
<tr><td><s>8 – 9</s>
8 – 10</td><td>Analysis of TextEditor and GEF/GMF
1 week Easter holidays</td><td>Analysis & Implementation of TextEditor and GEF/GMF
1 week Easter holidays</td></tr>
</table>
</td></tr>
</table>

	~~10 – 12~~ 11 – 13 | Implementation of TextEditor and GEF/GMF | Extension & Refactoring of TextEditor and GEF/GMF ~~13 – 14~~ 14 | Refactoring & Advanced Topics | Refactoring & Advanced Topics 15 – 17 | Finalization & Documentation | Finalization & Documentation **Table 6: Planning changes in weekly status meeting (14/04/2009)**

Goals for next one and a half week	• For Tuesday 21/04/2009 we defined the following goals:
	  ○ Create a simple domain model with person and location
	  ○ Implement a TextEditor using statically instantiated domain model for the highlighting
	  ○ Implement respectively generate a simple GMF editor which uses the domain model and allows to draw persons and its locations
	  ○ Additionally we should check if an automatic generation of the GMF editor with ANT is possible.
	• For Friday 24/04/2009 we defined the following goals:
	  ○ Link the implementation of the TextEditor with the one of the GMF editor working both on the same domain model.

28 Meeting Information 24/04/2009 & 25/04/2009

Date: 24/04/2009 & 25/04/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: HSR 1.258
Start: 10:00 am
End: 10:30 am
Keeper of the minutes: Tobias Forster

28.1 Agenda items

<table>
<tr><th>Item</th><th>Assignment / Information</th></tr>
<tr><td>Working progress</td><td>

We explained our working progress with the implementation of a tutorial application using the TextEditor and a GEF diagram editor. Both were separated until then and we started to link and synchronize them.
We also discussed about some details for the synchronization and why it is used. Also our ideas for the synchronization between the general domain model and the model for the graphical editor were interesting.
Additionally we reported the result of our analysis for the architecture of the TextEditor and GEF Editor synchronization. The different parts are focused on their core concerns and are therefore own plug-ins respectively bundles.

</td></tr>
<tr><td>Planning changes</td><td>

We adjusted the planning of the iterations and its core issues like shown in Table 7. The reason for this change was the higher research, analysis and implementation effort for the TextEditor, GEF/GMF and further RCP APIs as expected.

<table>
<tr><th>Weeks</th><th>Old topics</th><th>New topics</th></tr>
<tr><td>1 – 2</td><td>RCP Basics & Technology Know-How</td><td>RCP Basics & Technology Know-How</td></tr>
<tr><td>3 – 7</td><td>Advanced RCP, Spring DM & Automatic Build</td><td>Advanced RCP, Spring DM & Automatic Build</td></tr>
<tr><td>8 – 10</td><td>Analysis & Implementation of TextEditor and GEF/GMF
1 week Easter holidays</td><td>Analysis & Implementation of TextEditor and GEF/GMF
1 week Easter holidays</td></tr>
<tr><td>11 – 13</td><td>Extension & Refactoring of TextEditor and GEF/GMF</td><td>Extension & Refactoring of TextEditor and GEF/GMF</td></tr>
<tr><td><s>14</s>
14 – 15</td><td>Refactoring & Advanced Topics</td><td>Extension with further and advanced Topics</td></tr>
<tr><td><s>15 – 17</s>
16 – 17</td><td>Finalization & Documentation</td><td>Finalization & Documentation</td></tr>
</table>
Table 7: Planning changes in weekly status meeting (24/04/2009 & 25/04/2009)

</td></tr>
<tr><td>Goals for the next weeks</td><td>

For the next three weeks the following tasks are scheduled:

Refactoring of the TextEditor and Graphical Editor
Outsourcing of the domain model in an own service bundle
Synchronization between the TextEditor and Graphical Editor
Extension of the TextEditor and Graphical Editor with further functions
Documentation of the TextEditor, Graphical Editor and other used components

</td></tr>
</table>

29 Meeting Information 09/05/2009

Date: 09/05/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: HSR 1.269
Start: 03:00 pm
End: 03:45 pm
Keeper of the minutes: Ylli Sylejmani

29.1 Agenda items

<table>
<tr><th>Item</th><th>Assignment / Information</th></tr>
<tr><td>Working progress</td><td>

We explained our working progress with the implementation of a tutorial application using the TextEditor and a GEF diagram editor. The domain model was refactored and outsourced in an own service bundle. The TextEditor and the GEF diagram editor were implemented and refactored so far to be connected with the service layer. Spring was integrated for the wiring of the dependencies in the bundles. The next steps were to bring the bundles together to a whole application and the implementation of the synchronization between the editors and the domain model concerning the result of our analysis.
We also reported some problems we had with the menu items and commands during the implementation of further functions and extensions. Also, the further steps required to complete the implementation according to the timetable were discussed.

</td></tr>
<tr><td>Planning changes</td><td>

We adjusted the planning of the iterations and its core issues like shown in Table 8. The reason for this change was the higher implementation effort for the refactoring and the extension of the TextEditor and the GEF editors with further functions.

<table>
<tr><th>Weeks</th><th>Old topics</th><th>New topics</th></tr>
<tr><td>1 – 2</td><td>RCP Basics & Technology Know-How</td><td>RCP Basics & Technology Know-How</td></tr>
<tr><td>3 – 7</td><td>Advanced RCP, Spring DM & Automatic Build</td><td>Advanced RCP, Spring DM & Automatic Build</td></tr>
<tr><td>8 – 10</td><td>Analysis & Implementation of TextEditor and GEF/GMF
1 week Easter holidays</td><td>Analysis & Implementation of TextEditor and GEF/GMF
1 week Easter holidays</td></tr>
<tr><td>11 – 13</td><td>Extension & Refactoring of TextEditor and GEF/GMF</td><td>Extension & Refactoring of TextEditor and GEF/GMF</td></tr>
<tr><td>14 – 15</td><td>Extension with further and advanced Topics</td><td>Extension with further and advanced Topics</td></tr>
<tr><td>16 – 17</td><td>Finalization & Documentation</td><td>Finalization & Documentation</td></tr>
</table>
Table 8: Planning changes in weekly status meeting (09/05/2009)
</td></tr>
<tr><td>Goals for the next weeks</td><td>

The milestone M3 is set on the 22/05/2009 and covers the running application with TextEditor, text parser, GEF editors and synchronization including the appropriate documentation of these topics. Also, the abstract for the Java/Eclipse Magazine article, has to be finished and sent until M3.
For the meeting next week the running application is planned to be completed and demonstrated.

</td></tr>
</table>

30 Meeting Information 15/05/2009

Date: 15/05/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Container
Start: 01:00 pm
End: 02:30 pm
Keeper of the minutes: Ylli Sylejmani

30.1 Agenda items

Item	Assignment / Information
Working progress	• We presented the actual working progress and showed the running version of the Person Locator application. • The domain model and the implementation of the application were reviewed and discussed. Some additional tasks for the next refactoring iteration were derived. • We described the actual progress of the documentation and made estimation for the time required for the further documentation of the technical part.
Extension of the Person Locator application	• During the demonstration of the running Person Locator application a possible improvement was recognized. The creation of attributes could be dynamic instead of using pre-defined attribute definitions. The user would be able to specify custom attributes, which then are available in the context menu in the text editor. The description of a possible solution for such an extension should be described in the documentation.
Project planning	• There were no planning changes scheduled during this meeting. Except of some additional tasks for the next refactoring iteration, which are described below.
Additional tasks in the refactoring iteration	• The new Q4E plug-in for Eclipse should be installed and integrated for all Person Locator projects. • Currently the domain model data is not persistent. As a next step in the next refactoring implementation the dictionary data should be persisted in an XML document. • There are some cases, where fix strings are used in the source code. These strings should be externalized in a property file by using the “externalize strings” Eclipse feature.

31 Meeting Information 27/05/2009

Date: 27/05/2009
Attendee: René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: Phone meeting
Start: 05:05 pm
End: 05:20 pm
Keeper of the minutes: Tobias Forster

31.1 Agenda items

Item	Assignment / Information
Working progress	• We explained the actual working progress of the last one and a half weeks. • The documentation of the editors could be terminated as well as for further components and subjects treated. • We also mentioned our progress of the refactoring and extension steps of the Person Locator application. This covers the replacement of the Q4E plug-in through the new IAM as well as the integration of the Problems View and the Common Navigator Framework.
PDE Build problem	• We discussed additionally a problem, which occurred during the development steps. For unknown reasons Eclipse has problems to read some of the generated Manifest files. Using the legacy formatter solves the problem. Unfortunately this has also impact on the PDE Build, which seems to use the same parsing mechanism. Therefore the Headless PDE Build fails if such a Manifest file exists.

32 Meeting Information 05/06/2009

Date: 05/06/2009
Attendee: Wolfgang Giersche, René Eggenschwiler, Tobias Forster, Ylli Sylejmani
Location: 1.215
Start: 03:00 pm
End: 04:15 pm
Keeper of the minutes: Ylli Sylejmani

32.1 Agenda items

Item	Assignment / Information
Working progress	• We presented the actual working progress of the last two weeks. • Concerning the documentation of the thesis, we could close quite a number of tasks and are now working on last modifications on it for the finalization. • The abstract and the conceptual formulation of the bachelor thesis are finished and checked-in in the SVN repository. These have to be cross-read and approved by the examinator and the coach.
Setup of the reference architecture	• The examinator was also attendant of the meeting so we could setup and demonstrate the reference architecture on his computer. • There were no problems during the installation and the automatic build as well as the person locator example application worked at the first attempt.
Appointments	• Until Monday, the 08. June 2009, the abstract and the conceptual formulation of the bachelor thesis have to be cross-read so that we can make corrections if required until deadline. The abstract has to be delivered until Wednesday, 10. June 2009. • The deadline for the delivery of the bachelor thesis is Friday, the 12. June 2009, 12:00am.