HSR – University of Applied Sciences Rapperswil

Institute for Software

Introduce PImpl Refactoring for the CDT-Eclipse Plugin

Semester Thesis: Fall Term 2009

Andrea Berweger	Matthias Indermühle	Roger Knöpfel
aberwege@hsr.ch	mindermu@hsr.ch	rknoepfe@hsr.ch

Supervised by Prof. Peter Sommerlad

Version: December 18, 2009

Abstract
CDT is a plug-in for the Eclipse IDE to write programs in C and C++. Eclipse offers refactorings, functions that restructure the code without changing its functionality. We implemented the Introduce PImpl Idom refactoring for C++.
In C++ classes are usually divided in header files holding the declarations and source files holding the actual member function implementations. From the outside just public members are usable, but private members are also declared in the header file which has to be included to use the class. Introducing a new private member or modifying the signature of an existing leads to a change in the header and therefore forces all using classes to recompile. This is not a problem in a smaller project, but can have a great impact in larger projects that might take hours to recompile
The PImpl Idiom describes a way to avoid that. The idea is to introduce a new implementation class within the source file that holds all member functions. The original class is reduced to public members presenting a stable interface and a private pointer to an instance of the new class (the pointer to implementation = PImpl). Those public members use the pointer to delegate all calls to the corresponding member in the implementation class. Because this class is only known within the source file, changes to it do not require classes using the original class to re-compile.
But not only the re-compilation time drops, extracting the implementation in a separate class allows to use the original class as stable binary API and to hide the implementation in a library. With our refactoring all these advantages are just one click away.

Contents

1. Introduction

1.1. What is refactoring?

> "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure." [Fowler99]

Code refactoring means to to change the structure of source code without changing it's external functionality or behavior. There are multiple goals of refactoring code such as [Fowler05]:

- Improve readability
- Simplify code
- Improve maintainability
- Improve performance
- Improve extensibility
- Change code to a given programming pattern

During coding every one does refactoring, even if he or she is not aware of it. Already renaming a variable to a more suggestive name is a form of refactoring. But doing so by hand is very error prone. One easily forgets a variable at one point or makes a typo, so the code isn't compiling anymore.
With automated refactorings you avoid these errors and little mistakes. That's not the only advantage, speed is an other one. The computer is way faster in rewriting code than you'll ever be by hand.

1.1.1. Refactoring toolchain

Refactoring isn't just rewriting code manually or with help of an IDE. One should never forget that you need to have at least a code history in your version control system and unit tests to verify that the refactored code still does the same thing as before. Optional, but recommended, there should be used an automated build server. So you see immediately if the code you checked in is faulty and over time keep track about how often that happened.

1.2. Situation

Refactoring for C++ is a difficult task because of the language's complexity compared to Java. Features like preprocessor statements, macros or templates lay in the way of an easy automated refactoring.

1.2.1. Eclipse CDT

Eclipse CDT (C/C++ Development Tools) is a integrated development environment (IDE) for C++. It's base is the Eclipse IDE, a multi language software development environment and platform for other applications. CDT is written as a plug-in for this platform. Not as stable and high-performant as his Java counterpart, the Java Development Tools (JDT) , it's already a good C++ IDE.

1.2.2. Refactoring in Eclipse CDT

Compared to the number of refactorings in the JDT , CDT hasn't yet much to offer. To enhance this is the goal of the "Institut für Software" HSR Rapperswil. They introduced several new refactorings, for example:

- Extract Method ([Fowler99], S.110)
- Replace Number ([Fowler99], S.204)
- Move Field / Move Method ([Fowler99], S.142/146)
- Extract Subclass ([Fowler99], S.330)

For our work more important was the creation of a framework for new C++ refactorings. We didn't have to write strings and make changes direct in the file, since we were able to modify the Abstract Syntax Tree (AST) and use the refactoring source code unit written at the HSR.

1.3. Goals

Our intention was to make a more complex refactoring that is not part of the well known Fowler [Fowler99] refactorings. It's a specific one for C++ called "Pointer to implementation"-Idiom (PImpl).
The steps to reach that goal where:

- Analyze the PImpl Idiom on variants and special cases
- Learn to use the environment that Eclipse CDT provides
- Implement first version without variants and special cases
- Extend the first version to a fully functional refactoring
- Write the documentation to the work

1.3.1. Limits

C++ is such a complex and flexible language that one will always be successful in finding a case that does not work with our refactoring. So one of our first task was to decide what situations we should cover and what we left aside. But not only on the technical side the PImpl idiom doesn't work in all cases, it is not always meaningful to "PImpl" a class, for example when someone inherits from this class then it would be counter productive, since children could not use the implementation class.

1.4. Introduce PImpl Refactoring

1.4.1. Motivation

Listing 1.1: Dependencies of header and implementation files

```
Main.cpp -> A.h
A.cpp -> A.h
```

In C++ you have your classes usually divided in header files with the declarations and the source file with the member function implementations. From the outside just public members are usable, but private members are visible too in the source file of the header. If you introduce a new private member or change the signature of such a member you have to recompile all code using your original class. This is not a problem in a smaller project, but can have a great impact in larger projects that might take hours to recompile.
But not only the re-compilation time drops, extracting the implementation in a separate class allows to use the original class as stable binary API and to hide the implementation in a library. With our refactoring all these advantages are just one click away.

1.4.2. Execution

The "Introduce PImpl" refactoring creates a a new implementation class in the source file where all members get placed. In the original class only the public members are left and one private pointer (the pointer to implementation = PImpl) is added. All public members use this pointer to call their corresponding member in the implementation class. Because that class is only known in the implementation file, changes to it don't require to re-compile classes using the original class. (See 2.1 and 2.6)

2. The PImpl idiom

The Pointer to Implementation (PImpl) Idiom is also known as "Opaque pointer", "Compiler firewall idiom" or "Cheshire Cat" [WikiPImpl09].

2.1. PImpl in short

The PImpl idiom describes a simple technique to reduce coupling in large software projects by separating the interface from the actual implementation. A client class will only learn the interface of a facade class which is delegating all requests to an implementation class not visible to the client.

The following example shows, how the PImpl idiom is introduced to a very simple class.

Listing 2.1: Mainmember of the programm using the Example class

```
#include "Example.h"
int main (){
Example ex;
ex.foo();
}
```

Listing 2.2: Original Example.h

```
class Example {
public:
    void foo();
private:
   int getNumber();
};
```

Listing 2.3: Original Example.cpp

```
#include "Example.h"
void Example::foo() {
    int i = getNumber();
}
int Example::getNumber(){
   return 3;
}
```

Listing 2.4: Changed Example.h

```
class Example {
public:
    void foo();
private:
   int getNumber();
   const int number = 3;
};
```

Listing 2.5: Changed Example.cpp

```
#include "Example.h"
void Example::foo() {
    int i = getNumber();
}
int Example::getNumber(){
   return number;
}
```

Now we extract the constant **3** to a private const int member, the header changes and therefore the Main.cpp has to re-compile even though nothing Main.cpp actually uses has changed. The "PImpl Idiom" obscures these private changes by hiding the actual implementation in a separate class.
In order to do so it first generates a new class inside the implementation file, the so called implementation class (implClass). Afterwards it copies all members into the new class and delete the private ones from the original class. A pointer to the implClass is added that is now used by the public members to call their implementations in the implClass.
If there is no constructor yet, it has to introduce one and initialize the pointer. Otherwise it moves the implementation of these constructors into the implClass, adapting the call parameter and initializes the pointer with a call of its copied implementation. The destructor on the other hand doesn't have to call its counterpart in the implClass since he is automatically called when delete is called on the pointer. He just has to call that delete function to get rid of the object.

Listing 2.6: Refactored Example.h

```
class Example {
public:
    Example();
    ~Example();

    void foo();

private:
    struct ExampleImpl * _impl;
};
```

Listing 2.7: Refactored Example.cpp

```
#include "Example.h"

struct ExampleImpl {
public:
    void foo(){
        getNumber();
    }
    int getNumber(){
      return 3;
   }
};


Example::Example(): _impl(new ExampleImpl) {
}

Example::~Example() {
    delete _impl;
}

void Example::foo() {
    _impl->foo();
}
```

Now we can extract the constant in its own field without bein forced to recompile the Main.cpp again since Example.h doesn't change at all.

Listing 2.8: Refactored and changed Example.cpp

```
#include "Example.h"

struct ExampleImpl {
public:
    void foo(){
        getNumber();
    }
    int getNumber(){
      return number;
   }
   const int number = 3;
};

Example::Example(): _impl(new ExampleImpl) {
}

Example::~Example() {
    delete _impl;
}

void Example::foo() {
    _impl->foo();
}
```

2.2. What's the underlying problem?

Programs written in C++ consist of multiple classes, each usually implemented within two different files, a header and a implementation file. The header holds the declaration of the class and all it's members while the implementation file deals with the actual implementation.
The idea is, that a class using an other only imports the header and learns everything it needs without caring about the implementation. Thus bringing up the advantage of changes in the implementation file don't bother any users of the class.
However, this only works if no new members get introduced and no declarations of existing members get changed. If a new member is introduced to a class, even if it's private, all clients using this class will have to re-compile, although the actual interface hasn't changed at all.

2.3. Origin

First the PImpl idiom was introduced in the book "Advanced C++" [Coplien92] as "Handle Class Idiom". There it was used as a data storage for copies of the same object. It was implemented as shallow copy with a reference counter to determine when to delete the Implementation class.

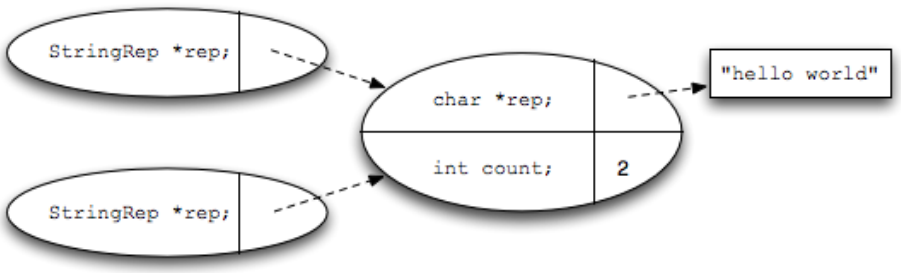


Figure 2.1.: Handle Class Strings with Reference Counting

2.4. The idea

The basic idea is, to separate the interface - meaning all public members - from the actual implementation.
To achieve this, a new class is introduced within the implementation file. This class will deal with the implementation and gets a simple forward declaration in the header file. It holds all members of the original class.
In the original class all private members are being replaced by a single pointer to the newly introduced implementation class. The public members get stripped of their code and will simply delegate all calls to the linked class.

2.5. Variations

2.5.1. Pointers

The simplest way is to just use the standard pointers offered by C++. However, those "naked" pointers have some disadvantages. Most important, they force the user to properly free the memory on the heap after a referred object isn't used anymore. In our case this means we need a destructor to actively delete the implementation object.
To circumvent this problem, the boost and TR1 libraries offer a shared pointer. Those are intelligent pointers, that count the number of references to an object and delete it when the last reference gets deleted.
We decided to offer those three possibilities to the user in the wizard.

2.5.2. Copy Problem

To understand the copy problem, it's needed to know how C++ reacts when requested to make a copy of an object.

Listing 2.9: Copy problem

```
class Example
{
public:
   Example(): number(new int())
   {
   }
   void setNumber(int a)
   {
      *number  =  a;
   }
   int getNumber()
   {
      return *number;
   }
private:
   int * number;
};
int main() {
   Example ex1;
   ex1.setNumber(1);
   Example ex2 = ex1;
   ex2.setNumber(2);
   std::cout << ex1.getNumber() << std::endl;
   std::cout << ex2.getNumber() << std::endl;
}
```

Because no copy constructor is specified in the class "Example", C++ uses a standard copy constructor, just copying the values of all members to the new object. As "number" is a pointer to an integer, its value is the memory address of the actual integer and "number" in the copy object will point to the same integer instance. Therefore this code would produce the following output:

Listing 2.10: Undesired output

```
2
2
```

To avoid this behavior a copy constructor is needed to create a new instance of integer.

Listing 2.11: Possible copy constructor

```
Example(const Example & oldObject)
: number(new int(*oldObject.number))
{
}
```

This copy constructor instantiates a new object for "number" to point to, calling the copy constructor of integer. Our sample code now produces the following output.

Listing 2.12: Desired output

```
1
2
```

Because the entire idea of PImpl is build around a pointer, the copy problem needs special attention. There are three different possibilities selectable by the user to handle this.

2.5.2.1. Deep copy

A copy constructor is inserted to the original class that calls the copy constructor of the implementation class. Therefore the copy behavior of the refactored class is exactly the same as it was before and fulfils the requirement, that a refactoring shouldn't change the external behavior.

2.5.2.2. Shallow copy

If the user explicitly wishes his class to have a copy problem, meaning the original and the copy would share the same implementation object, he can select the shallow copy option and no copy constructor is inserted. This option is only offered, when a shared pointer is selected, as many objects pointing to the same implementation would bring up the problem when to delete it.
This behavior is often appreciated when the class has just computing members and stores no values or pointers.

2.5.2.3. Prevent copying

To completly prevent any other class to make a copy, the copy constructor can simply be declared to be private. The user can request the refactoring to do this explicitly or to make the class heir from boost::noncopyable. Any try to copy an object from an other class would then lead to a compile time error.

2.5.3. Handling static members

The presence of static members causes some serious problems when implementing the PImpl idiom. Should they also be transferred to the implementation class, or should the stay within the original?
Public static member fields need to stay in the original, otherwise they wouldn't be visible from outside and using public static constants is quite common.
Public static member methods therefore also need to stay within the original class to have unchanged access the public static member fields.
This implies private members to also stay in the original class, as a public static member method could need access to them.
We therefore decided to leave everything that's static in the original class.

2.6. Advantages

Separating the declaration and the implementation of the function members in separate classes has several advantages. By holding the entire implementation in a separate class within the source file, it can easily be changed without having to recompile all the classes that use this class. This reduces building time significantly in large software projects, especially if it concerns often used classes.
The separation allows us to hide the implementation of our features in a library with the front class as stable binary API. That allows us to change the internal of the class without affecting the original software at all.

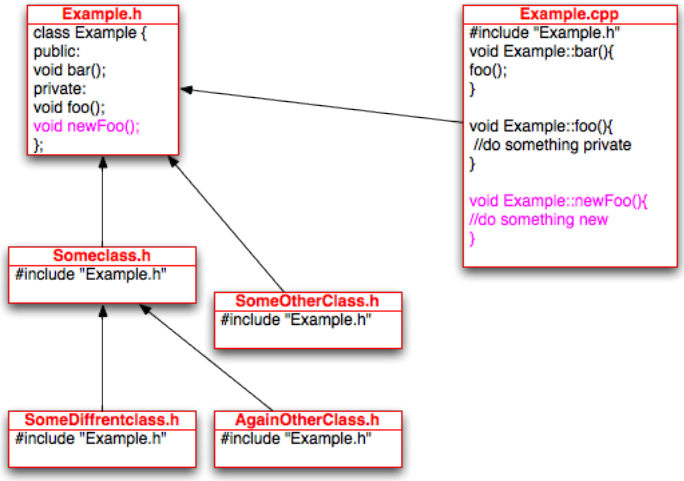


Figure 2.2.: Without PImpl: Everything needs to recompile

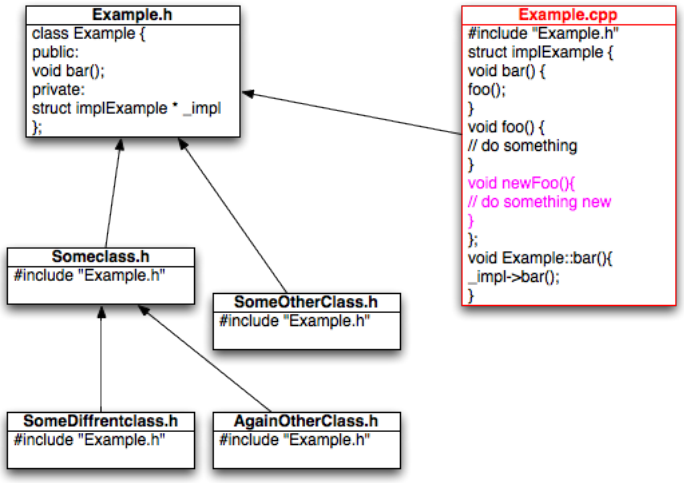


Figure 2.3.: With PImpl: Only "Example.cpp" needs to recompile

2.7. Disadvantages

The additional indirection reduces the runtime speed of a member call.

The constructor needs more runtime since the implementation object has always to be allocated on the heap which can be expensive, especially when multithreading calls for synchronization.

The PImpl idiom can't implement everything.

- It is not suitable for any class having a base class in most cases, because the base class and its members would be invisible to the implementation.
- Public fields can't be visible for both, the client class and the implementation class since the implementation class has no reference to the original class. There is a way around that problem by giving the implementation class a backpointer to the original class. But that means to introduce a cyclic dependency which is not favorable.
- Static members need to be fully called (Example:: staticMember) from inside the implementation class.

3. The Eclipse Refactoring Framework

3.1. Plugin development

3.1.1. Introduction to plugin development

This is just a short introduction to plugin development in eclipse, because the development of plugin's is fully supported by the eclipse PDE (Plugin Development Environment). So it's quiet easy to create plugin's, but there are one or two problems, I will refer to them. The big part of the plugin can be configured in plugin.xml, which file will be opened in a special editor.

3.1.2. Dependencies

Dependencies allows you to refer to other plugin's in your workspace and/or libraries. If your dependencies addict to version's, it's can be tricky that this plugin runs in an other eclipse configuration. First, the other eclipse configuration have to have this dependend plugin's and second, they have to be in the minimal version used at the export of your plugin. Otherwise your plugin wouldn't be loaded from eclipse.

3.1.3. Extensions

In plugin development you can extend the environment, such as menus, editors, view and so on. To do this you can define extensions. This extension only works correct, if the structur of your extension point precisely refer to the extension point which you want extend. Otherwise your extension will not be shown in the environment or it creates a mess, like unsorted menus or contextmenus.

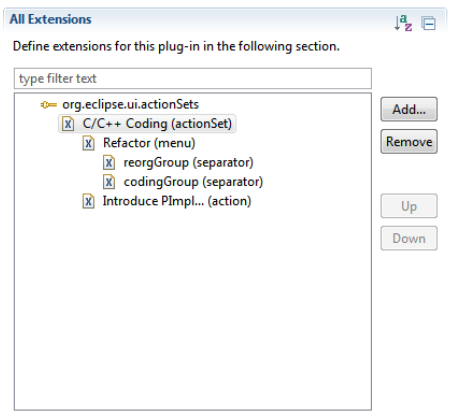


Figure 3.1.: Introduce PImpl extensions

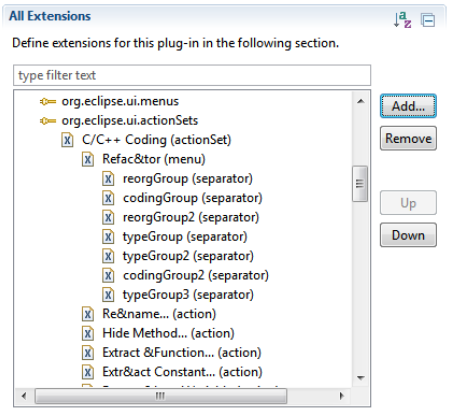


Figure 3.2.: CDT extensions

3.1.4. Export plugin

To export a plugin you have to create a build.properties file, otherwise the plugin will not be exported correctly. The PDE support's to automatically create this file. In the overview register of the plugin.xml under exporting is a link "Build Configuration", which automatically creates the required file, and expand a new register (Build) in plugin.xml. This register have to be configured, so the PDE can export the configured sources. It is very important that a library and a folder is added to the list on top, which defines the order of building these sources.

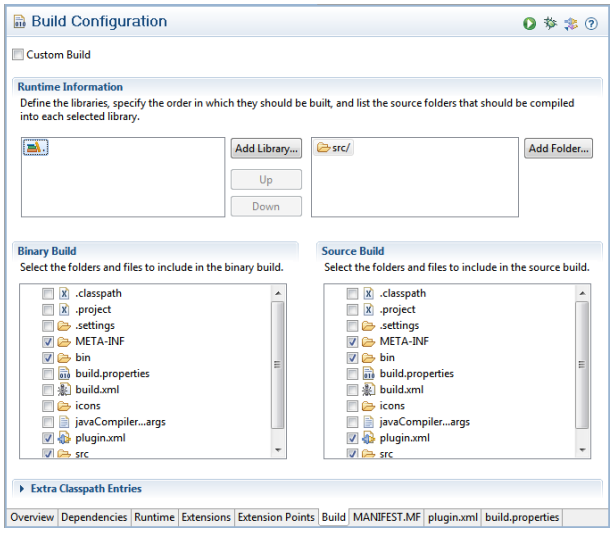


Figure 3.3.: build.properties in eclipse PDE editor

To create a runnable plugin, it's nessecary to create a feature project. This project only contains a list which projects have to be exported. The newer version of eclipse don't accept it to copy new plugins in the plugin folder. In the plugin folder is managed by eclipse and only plugins will be load, which are installed by eclipse. Other plugins, which only can be droped in have to be in the dropin folder or in the feature folder. The feature folder additionally needs a feature plugin which contains all plugins to load and generates the correct folder structure. I'm sure about the difference of the dropin and feature folder.

3.2. Changing the code

3.2.1. The AST

The abstract syntax tree (AST) is a tree structure representing the actual written code in a file. All transformations made by a refactoring are first applied to the AST and then parsed back to literal code.
The root node is called translation unit, meaning the entire file. Children of the root are typically compiler instructions, followed by one or several declarations. Those declarations have further children, specifying attributes like names and the declarations of other objects encapsulated by them.

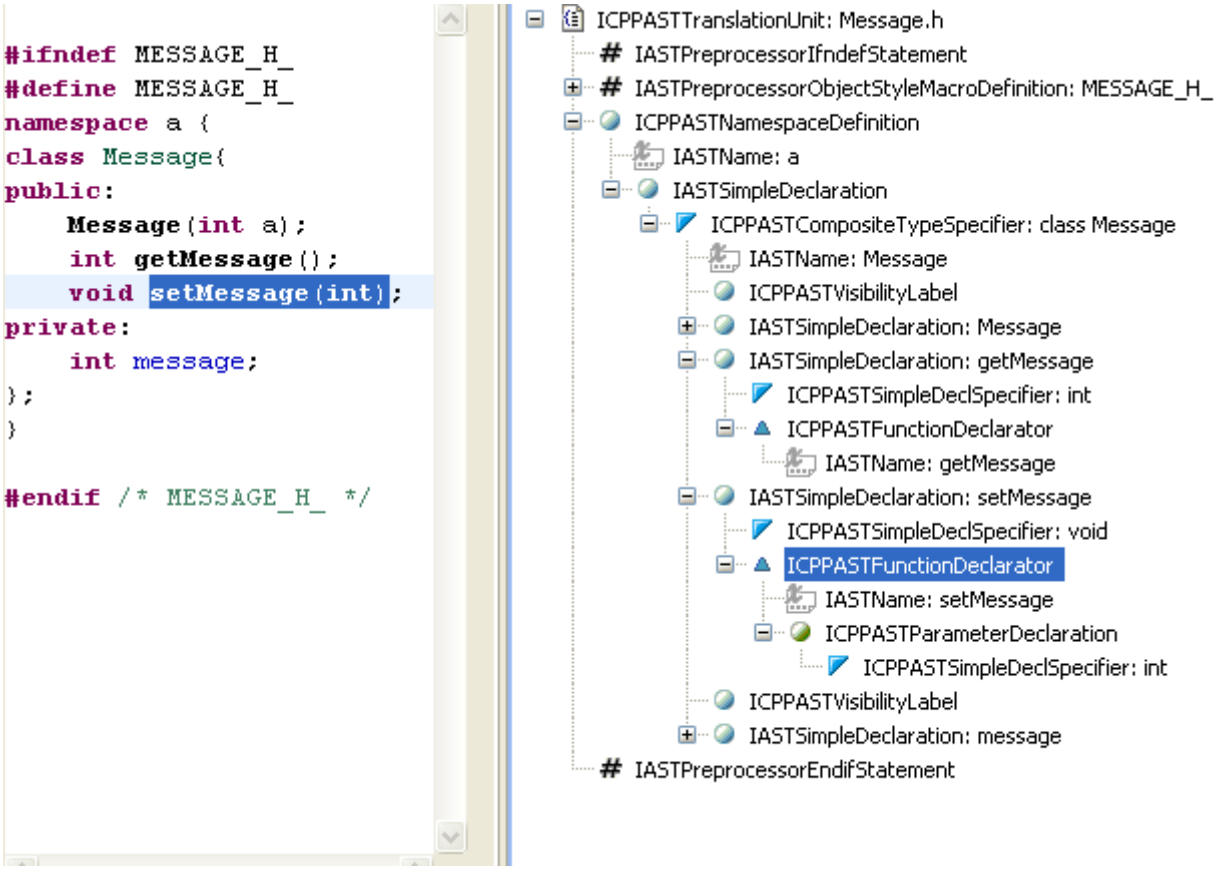


Figure 3.4.: Code example and its AST representation

3.2.2. Rewriting the AST

Rewriting source code means rewriting the AST. Every statement is represented there, so every change comes down to adding, removing or replacing nodes. This job is done by ASTRewrites provided by the refactoring framework.
Every ASTRewrite has a root node and can modify any descendants of it. The three most important method calls are insertBefore, replace and remove.

Listing 3.1: Interface of ASTRewrite

```
ASTRewrite subRewrite = aRewrite.insertBefore
   (parent, insertionPoint, newNode, editGroup)
ASTRewrite subRewrite = aRewrite.replace
   (node, replacement, editGroup)
aRewrite.remove(node, editGroup)
```

It is important to know, that those modifications are not applied as soon as they get called. ASTRewrite stores the modifications and rebuilds the entire AST later. This has some impact on how it can be used.
In the call of insertBefore, there is a "parent" handed over, this specifies the node in the AST, that our newly inserted "node" should be a child of. The parameter "insertionPoint"

defines the exact position between all children of "parent" where "node" should take place. As these parameters are nodes actually present in the AST, what happens, if one modification removes node "A" and an other modification wants to insert node "B" before node "A"?
It is therefore critical, to make sure all referenced nodes will stay, otherwise null exceptions appear.
A once inserted node and its descendants or a node replacing another can later be modified using the returned subrewriter.

3.3. Testing the refactoring

Testing a refactoring is difficult, as it mainly consists of a single big class. As this class is designed to be invoked by a refactoring process, it is not possible to make simple unit tests checking the interfaces. Fortunately, there is a comfortable framework for testing C++ refactorings [GrafButtiker06], based on JUnit.

3.3.1. CRefactoring

The abstract class "CRefactoring" takes a test file, builds up the entire environment and runs the actual tests specified in the concrete implementation class. This means:

- Starting a CDT build containing the refactoring
- Creating a new project with files generated from the test file
- Compiling the project using the toolchain of Eclipse
- Running the tests

3.3.2. IntroducePImplRefactoringTest

This class basically overrides two methods of "CRefactoring".

configureRefactoring() reads properties specified in the test file to configure the parameters of the refactoring.

run() starts the refactoring and checks the success of the condition checks. It finally compares the refactored code with the expected result from the test file.

3.3.3. The test file

"CRefactoring" is designed to interpret a text file with suffix "rts". In this file, tests can comfortably be designed by specifying the original state of the files containing the target class, the desired configuration of the refactoring and the expected state after a successful refactoring. It uses markers for the sections.

//!	The name of the test, displayed in the JUnit monitor.
//@.config	Section to define properties, used to configure the refactoring
//@<filename>	The initial state of a file, affected by the refactoring
//=	The expected state of the previous file after the refactoring is done

Within a //@ section a code element can be surrounded by /$/<an element>/$$/ to mark it as selected by the user when the refactoring starts.

3.3.4. Problems

The final check performed to assure the output is correct compares the expected code with the actually refactored code based on strings. This is fine for refactorings just manipulating small parts of code, but has some serious drawbacks on a refactoring manipulating entire classes like "Introduce PImpl" because semantically identical code can easily fail a character-by-character comparison.
In consequence, we very often had to adept the tests to match the code instead of vice versa as it's supposed to be in test driven design.

3.3.4.1. Whitespaces

How the literal code is finally generated from the AST is not transparent to the developer and depends on how and when nodes are manipulated. Tabs, new lines or even spaces are added outside the control of the refactoring class. So we had to hunt down spaces and correct line breaks after many code manipulations.

3.3.4.2. Ordering

In which sequence members are declared or defined is not important to the functionality of the code, it's more a matter of style. To complete a test, we sometimes had to rearrange the sequence of definitions or declarations.

3.3.4.3. Comparing nodes

To consequently check the function of our "NodeFactory" class, a possibility to check equality of nodes would be useful. Unfortunately, we ran out of time to add this functionality.

4. Description of the refactoring code

4.1. Refactoring Runner

The RefactoringRunner is a class, which extens the the RefactoringRunner-class and initializes the refactoring process. But first we have to know how this class will be called from Eclipse. The extensions from the plugin are responsable to call the right classes, but there are some middle men. Extensions of type actionSet creates new menu entries. These menu entries refer to a delegate classes, which implements the IWorkbenchWindowActionDelegate interface. Eclipse calls the run method of this interface, if the user choose these menu entry. This run method evaluates from where the call came, this means from which active part of the workbench, also known as views, the call was sent. As example, the PImpl refactoring only respond if the project explorer, the outline or the editor is active. Otherwise the call is blocked an not send to the next instance.

If one of these view is activ, the action delegate collects some information depending on from where the call was sent and creates an new instance of an action class, which extends the RefactoringAction class. Also this interface contains a run method, which will be called from the delegate. This action class also collects some information like selection, file, etc. Not till then this run methods calls the RefactroingRunner with the collected data.

Now, the RefactoringRunner initializes all necessary classes to start the refactoring process. First, it initializes an info class. This is something like a data transfer object (DTO). This DTO stores all information, which has to be transfered from the wizard to the refactoring class. The next step is to initialize the refactoring object. This object processes the refactoring to a later time. The runner also needs to instantiate a wizard, which interacts with the user. Last but not least it's necessary to instantiate an operator. The operator is the controlling instance for the refactoring. It starts the refactoring and controls the whole process.

Listing 4.1: IntroducePImplRefactoringRunner

```
public class IntroducePImplRefactoringRunner
   extends RefactoringRunner {

public IntroducePImplRefactoringRunner(IFile file,
      ISelection selection,   ICElement element,
      IShellProvider shellProvider) {
   super(file, selection, element, shellProvider);
}

@Override
public void run() {
```

```
    IntroducePImplInformation info =
        new IntroducePImplInformation();
    CRefactoring refactoring =
        new IntroducePImplRefactoring(file, selection,
                            celement, info);
    IntroducePImplRefactoringWizard wizard =
        new IntroducePImplRefactoringWizard(refactoring, info);
    RefactoringWizardOpenOperation operator =
        new RefactoringWizardOpenOperation(wizard);

    try {
        refactoring.lockIndex();
        operator.run(shellProvider.getShell(),
                  refactoring.getName());
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } catch (CoreException e) {
        CUIPlugin.log(e);
    } finally {
        refactoring.unlockIndex();
    }
}
```

4.1.1. RefactoringWizardOpenOperation class

The operator initates the whole refactoring process and is started by the run method. It asynchronously calls method by method of the refactoring class and interoperates with the wizard dialog. After the operater is started, it calls the checkInitialconditions() of the refactoring class as first step. After this method is worked by the refactoring, the operator shows the wizard dialog dependent on the informations which were collected in the initial conditions. It shows page per page until the refactoring specific pages are complete. As next step, the operator calls the checkFinalConditions() method of the refactoring. Directly after this call it calls the collectModifications() method. If there are some errors or warnings in this method, it shows up a LTK based wizard page to display the occured problems to the user. If no errors occured and the user continues the refactoring, the operator shows the next LTK based page, which contains the conclusion of the affected changes. So the user can inspect the changes and if he's satisfied with, he can continue the refactoring, what means the operater create the changes on the files. At the bottom is an illustration of this process but note that the operator only calls the checkInitialConditions and the wizard directly. Between the operator and the methods checkFinalCondition and collectModifications are many other instances which supports the concurrency of the process. Explaining the whole process would end in a hugh expand of this documentation and is not important for the refactoring.

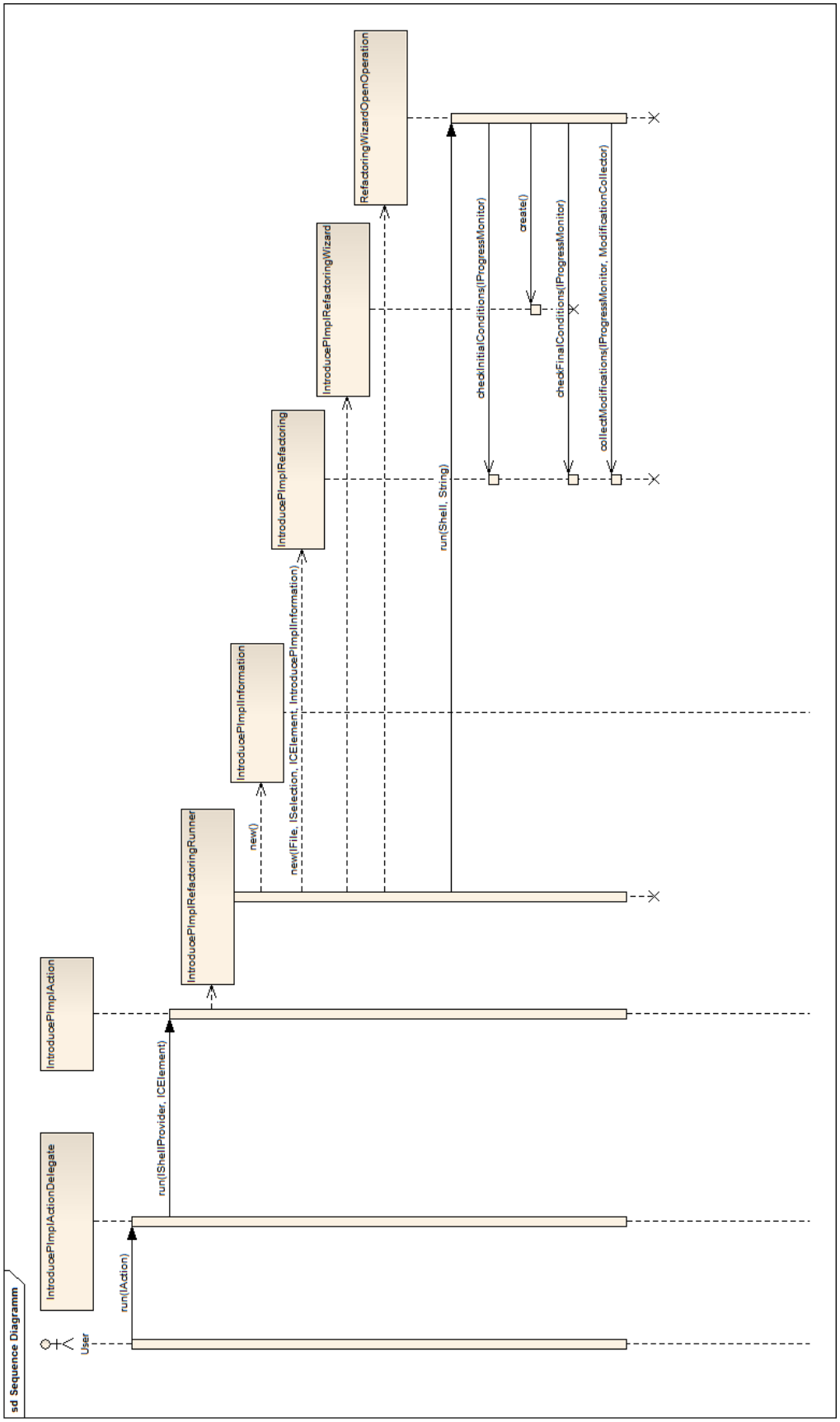


Figure 4.1.: Simplyfied process illustration

4.2. Refactoring Wizard

4.2.1. Page overview

In most cases, the refactoring has to interact with the user. So at this point, it's time to take a closer look to this class and how it works. The wizard attend the user through the refactoring process. The wizard contains at most four pages. All the allocated information in these pages will be collected by the wizard and provided with aid of the info class to the refactoring.

The last two pages will by created and provided by the LTK. These are the basic pages of each refactoring. One shows possible problems and warnings, the other the conclusion of the affected changes.

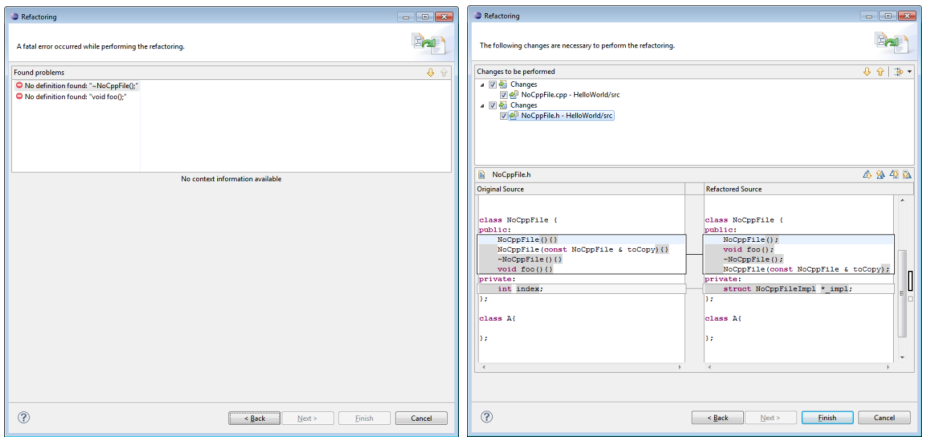


Figure 4.2.: Basic pages, optionally shown by all refactorings

The first page is optional and will only be shown, if the header file contains more than one class and the selection is not precisely enough to identify which class the user want's to apply the impl refactoring. So, on this page have the user the possibility to choose his preferred class.

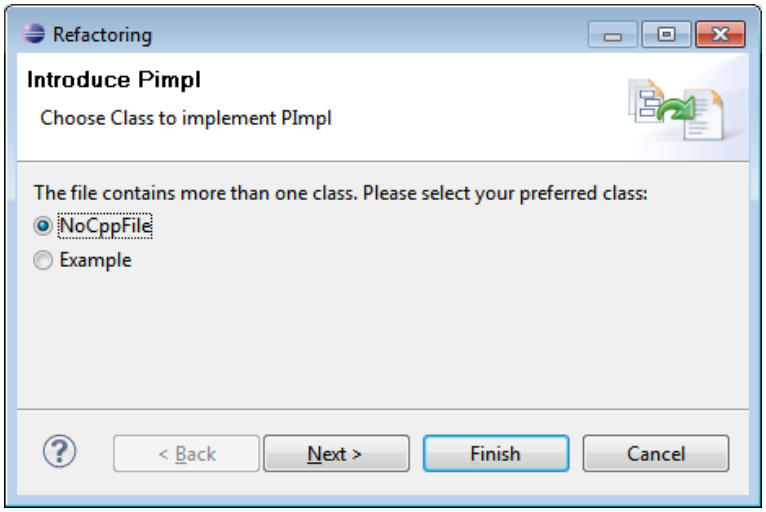


Figure 4.3.: Class can't be automatically precisely identified

The second page is the most important page, and it's also a mandatory page, which ever will be shown. On this page, the user can provide how the PImpl will be created. The following options can be choosen.

1. The name of the implementation class
2. The name of the pointer variable
3. The type of the implementation class (class or struct)
4. The type of the pointer (standard pointer, shared pointer with BOOST or TR1)
5. The type of how the class can copied (deep, shallow, nocopy, noncopyable (require BOOST))

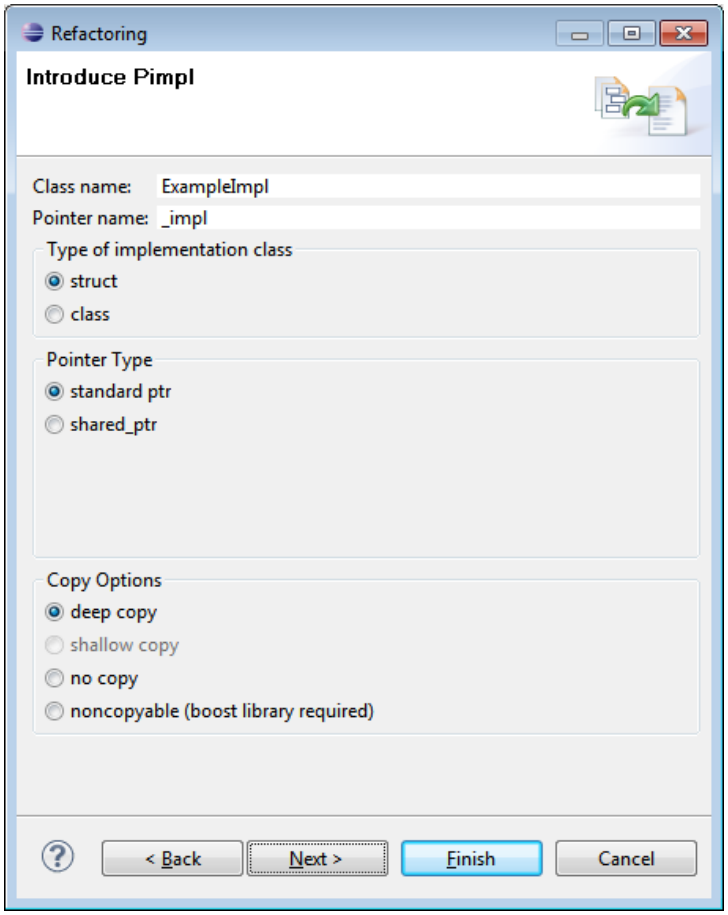


Figure 4.4.: PImpl refactoring needs informations

4.2.2. Technical background of the wizard class

The graphical user interface (GUI) have to be designed by the standard widget toolkit (SWT). The basic container which shows the title bar and the four buttons at the bottom of the dialog is provided by the LTK. This container also provides basic functions like previous- or next page, message presentation in the title bar and so on. This basic container is implemented as UserInputWizardPage class and all own designed pages have to extend this class.
The wizard page him self contains a method "addUserInputPages" in which all needed pages have to be instantiated. In this method, the inptu pages only be instantiated, but not created. UserInputWizardPage will provide a method "createControl", which will be called at a later time by the LTK. Not till then, the LTK creates the page.
As said the Pimpl refactoring contains two user input pages, whereof one is optional, dependent on the info class, which is partially initalized in check initial condition of the refactoring, which will be explained in the next section. The first, optional, page will only be added, if the informations of the info class contains more than one class and no class is identified. In this case, the wizard adds the page in which the user can identify his preferd class. Otherwise only the mandatory page will be added to collect the preferd options of the user to process the changes.

4.3. checkInitialConditions()

The function of this method is very simple. The only goal of this method is to find the selected class and it's associated header file in which the selection, respectively the cursor is. To accomplish, it's necessary to differ from where it was called.
If the call was out of a header file, of course, the method don't have to find the header. But in this case the method first have to find the selected node which is owned by the selected region. With aid of this node it have to find the associated class node. If a class node can be found it will be saved in the info class. If no class node can be found means, that the cursor position was outside of a class declaration. In this case the method will collect all classes in the file, which later would be displayed by the wizard.
This was the process if the refactoring was started from a header file. The process is a little bit different if the refactoring was started from a source file. The first step is the same as by the header file. The method needs to find the selected node which is owned by the selection. Because of it's a source file you can assume, that the selected node is a function definition. So the next step is to resolve the declaration of this function definition. On succeed of this resolving the method gets the header file and continous like with the header file. In this case it will find a class, because it's not possible that the resolved decclaration is not owned by a class.

4.4. checkFinalConditions()

This method collects the second part of the needed information, even the source unit. Certainly it's necessary to have a source file to create the PImpl refactoring. The process to find the source unit is not before the wizard, because of if the header contains more than one class and the selection is not set to a class, the user first have to choose his preferred class. So the wizard first must be shown to the user. After the user entered his preferred choice, the refactoring is able to find the correct source file.
But the almost first step is, that the method tries to find a definition for each declaration. If only one declaration don't have a definition the method will stop working and informs the user which declaration these are. If all declarations have a definition, as a next step, the method tries to find all source files of all definitions. It's possible, that the function declarations are splited on more than one file. There are three possible cases to differ:

- **More than one source file is found**
  At this time it's not supported to collect function definitions from more than one source file. In such a case the refactoring throws an error and informs the user which files are found. This function will may be realized as an own refactoring. In most cases the method will find exact one source file.

- **Excatly one file is found**
  This is the standard behavior and the only thing to do is to create a translation unit of this file.

- **No file is found**
  An other special case is that all function defnitions are inline in the header implemented. If so the method will no source file find. The actual reaction to such a case is, that the method tries to create a new file with the same name as the header file,

of course with the file extension ".cpp". If no such file exists it will be created by the method. If a file with this name exists, it get this existing file and inserts the changes to the end of this file. Note: The file create process is temporary implemented as workaround, because there actually exists a file create change, which creates a new file as a change at the end of the process if the user accepts the changes, but it don't works correctly still yet. So we had to do a workaround. The workaround is, that the file will be create before the user accepts the changes. If the user abort the process, the file is created and will not be removed. Note: To create a new file in a project, it's nessecary to use the IFile interface. This interface provides a create file method, which takes a InputStream parameter. This parameter don't have to be null. If this parameter is null, the file will be marked as not local, that means the file will be created in the project but not pyhiscally on the hard disk. It's nessecary to create an empty file stream. To prefer a ByteArrayStream with an empty String. So the file also will physically created on the hard disk.

4.5. collectModifications()

This method gets called when the user has chosen the parameters in the wizard and proceeds. It calculates all necessary changes to both the header and the source file. Those changes will be presented to the user to check on the next screen, before they actually get applied on the files.

4.5.1. The idea

As inserting nodes before other nodes that might change is dangerous, we decided to completely remove the original code and re-insert our new code node by node only at the last position of four different sections. That means it's critical not to change a node after it's once inserted, as it could compromise all following nodes. Those sections are:

1. The public section of the original class header
2. The private section of the original class header
3. The namespace of the original class in the source file or the entire source if none is specified
4. The implementation class

Because all declarations of a class are direct children of the same parent node, the public and the private section of the original class are technically the same, the only difference being the last previously visited visibility label. To keep the principle of only inserting nodes at the last position, private members are hold in a buffer and inserted at the end.

The implementation is quite straight forward. First the new nodes for the new original and the implementation class are created. Then we traverse all declarations of the original class, check their type and pass them to specific handlers, which modify the node and/or generate new nodes and insert them at the right place.

After all declarations have been taken care of, some PImpl specific members are added.

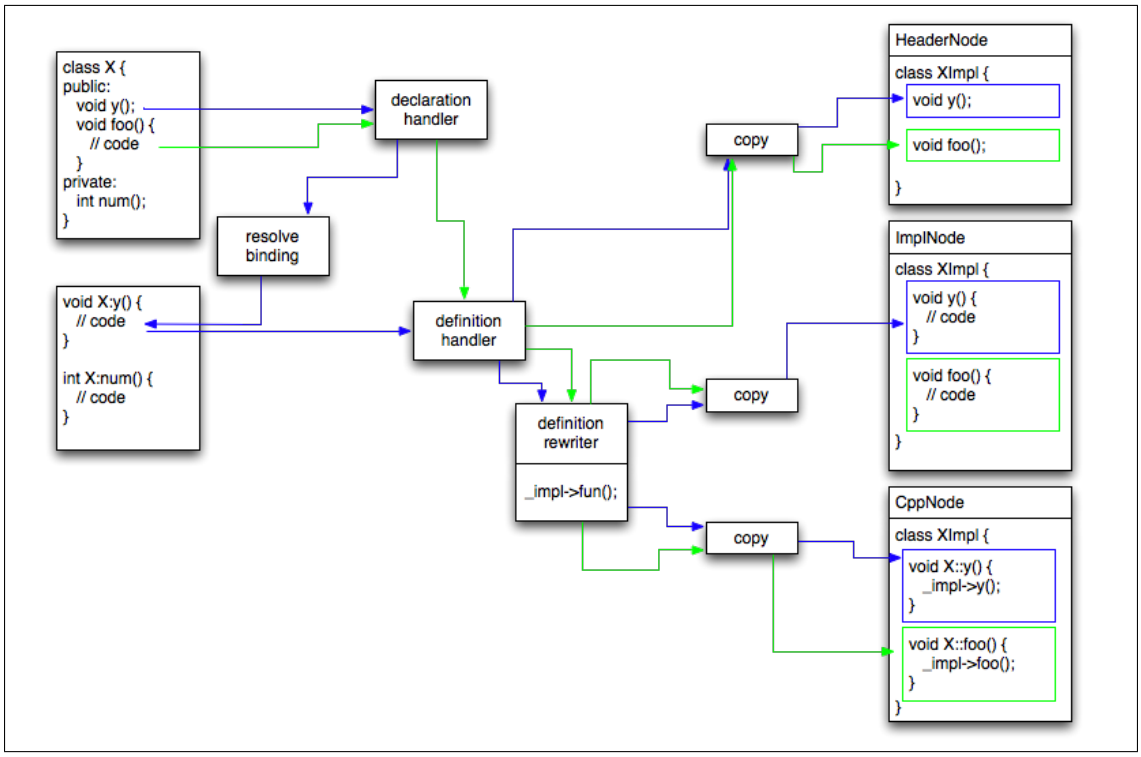


Figure 4.5.: The flow of processing Nodes (Green: Inline implementation)

- Basic constructor, if no constructor has been added already
- A destructor if needed
- Private or public copy constructor, if needed
- The pointer to implementation

At last, the buffered private static members are inserted into the header.

4.5.2. handleDeclarator()

A declarator represents a member field. As those don't need further definition, declarator ist just written into the implementation class or, when static, hand over to the static declarator handler.

4.5.3. handleStaticDeclarator()

We decided to put static members into the original class (see description of PImpl variants), therefore the handler checks wether the field is private and inserts it into the private or public section of the new original class accordingly.

4.5.4. handleFunctionDeclarator()

This very simple handler looks for the definition for the passed function declarator and passes a copy of it to the function definition handler, after removing the original from the source file.

4.5.5. handleFunctionDefinition()

A function definition can represent member functions with different roles that need to be differently handled, so this method checks for those roles and passes the definition to specialized handlers. We differentiate between:

- Constructors
- Copy constructors
- Destructors
- Static member functions
- "normal" member functions

4.5.5.1. handleConstructor

The definition is inserted into the implentation class after beeing modified to match its name. To allow an unchanged instantiation of the refactored class, the handler then writes an unmodified declaration into the new class header. The definition to be written into the source file, initiates the pointer with a call to the according constructor of the implementation class, passing all parameters.

4.5.5.2. handleCopyConstructor()

This definition is also inserted into the implementation class but in addition to adapting the name, the parameter type also has to be changed to match the implementation class. The copy constructor is not added to the original class, as the copy behavior is handled later.

4.5.5.3. handleImplDestructor()

Similiar to constructors, the name of the destructor needs to be adapted before inserting it into the implementation class.
The destructor is not inserted into the original class, as its necessity is decided later on.

4.5.5.4. handleStandardFunctionDefinition()

A standard function definition is inserted into the implementation class. To allow its usage from outside, an unmodified declaration is created and inserted to the new original class header. Its according definition is created to call the implementation method passing all arguments and eventually return the received value from it and added to the source file.

4.5.5.5. handleStaticFunctionDefinition()

As we decided to keep static members in the original class (see description of PImpl variants), an decalration for the definition is inserted into the new original class header and the defintion itself into the source file.

4.5.6. Extendibility

A big advantage of this concept is it makes it very simple to change the handling of certain types or specify new types and their handling. Simply add a new case into the decision tree in collectModification() and write a new handler.

4.5.7. Limitations

The handler concepte is high extendable, but yet, it have some limitations. Becauseof it needs a handler for each node type there may are some node types, which still are not implemented. Actually exists handler for declarations, function declarations, function definitions and visibility labels. Other nodes in a class declaration may are not identified. The result of this limitation is, that this unidentified nodes will be lost if the introduce PImpl refactoring will be applied. In future, may many other nodes will be designed, which each node needs a handler. Otherwise it will be ignored.

4.6. Helper Classes

We extracted functions that we thought could be used by other refactorings into separate helper classes.

4.6.1. NodeHelper

NodeHelper.java provides methods to check certain aspects of nodes form the AST. This class is fully static and provides methods which identify special nodes. This class is needed because of definitions like constructor, destructor and copy constructor are all of the type ICPPASTFunctionDefinition. A constructor is identified due to his declaration specifier, what means his return type, which is unspecified. Additionally the name have to be the same as the name of the class. The only difference to the destructor is, that the name starts with a tilde (~). The rest of the signature is the same. The copy constructor is analog to the constructor, only that the constructor has to have a parameter of the type of his own class and it have to be a reference pointer. About these characteristics handle the methods to identify the type.
The method "isFunctionDeclarator" is mainly useful because a declaration and a function declaration have nearly the same signature. It returns if a passed node is a declaration of a function. The method "isEmptyDeclarator" identifies a lonely standing semicolon. Semicolons can ba typed as much as you want and the code still will compile, but the AST handle they all as declaration nodes. The characteristics of such nodes are that the content is null or as unspecified declarated.Due to this, the method can identify this node types. The shortest and easiest method is the "isStatic". The only work this node does, is, to identify the passed node type and extract the static field of this node dependent on this type.
There are junit tests for all those Methods in the project introducepimpl.test under the class NodeHelperTest.

The CDT already contains helper classes in the refactoring utils. This package also contains already a NodeHelper class. The plan is, to merge this both classes together if the introduce pimpl refactoring will be integrated into the cdt.

4.6.2. NodeFactory

The NodeFactory class contains certain methods to build special nodes like constructors, destructors, declarations out of definitions and so on. The node factory only contains creater for function definitions, but there's an additional method called "createDeclarationOfDefinition", which will create the declaration of a definition. So, it's not needed to implement all creater twice. One for the definition and one for the declaration.
At this time, there is no test for this helper class. The reason for that is, that's not so easy to compare two nodes. First, it's nessecary to create a node helper method, which can compare two nodes. There are some comparer for nodes in the refactoring utils in the ui package, but these helpers are all specially for some node types. Acually there is no comparer, which can compare each types of nodes. Because of there are many node types, this method will be very tricky. One simple solution may be is to serialize both nodes and compare his file contents.

4.6.3. NodeContainer

The node container is a generic class to hold a node and his associated ASTRewrite together. Normally, you can get an ASTRewrite of a translation unit, if you use this rewrite to insert or replace a node you get a new ASTRewrite back. This ASTRewrite is generated for this inserted or removed node. If you have to do more changes on this node, you have to use this ASTRewrite. You can't use the ASTRewrite of the translation unit, because of the new node only is collected as a change. The node isn't a part of the translation unit yet, not till than the change will be performed at the end of the refactoring.
The node container supports to get sure, that the node and his associated ASTRewrite stay together. The node container additionally supports to do the rewrites directly on the container. That means the container also contains the "insertBefore(...)", "remove(...)" or "replace(...)" methods. These only are delegates to the internal rewrite of the node.
Note: The "insertBefore(...)" method contains less parameter than the original "insertBefore(...)" method of the ASTRewrite. The lost parameter is the parent node which is not necessary, because the parent is the node held in the container.

The return values of the manipulation methods of the node container are actually the new rewrites, which results out of the rewrite. May be it will be better to return new node container, which contains the inserted node and it's resulting rewrite.

4.7. collectModifications() - The first approach

4.7.1. Overview

This is the description of our first approach. The basic difference is the point of view. The idea was to simple map the steps a programmer would perform to modify his existing class to match the PImpl Idiom into program code, instead of reverse engineering what the ASTRewrite would have to do as it was done afterwards in the final solution.

4.7.2. Steps

1. Collect all members

2. Add includes
3. Create declaration of implementation class
4. Copy all members to implementation class
5. Remove any private members from original class
6. Declare pointer to implementation
7. Rebuild constructors in original class
8. Map public member functions to the implementation class

4.7.3. collectMembers()

The declarations of the class received by parameter get traversed looking for actual declarations and visibility labels. The method collectMember() creates a MemberContainer for each declaration, holding the nodes of the declaration and the definition and the rewriters for both of them. Based on the last visited visibility label, the container will be added to the collection of public or private members.

4.7.4. insertIncludes()

Some parameters selectable by the user require the inclusion of additional libraries, boost or tr1 in particular. Those are included as literal nodes in the header file.

4.7.5. createImplClass

In short, this method takes a copy of the class declaration in the header, renames it, changes the type to class or struct, replaces the declarations with the actual definitions the get an inline implementation and inserts it in the source file.
The copyDefinitionsToImpl() method also checks the members for being constructors or destructors, adapting their name and, in case of copy constructors, the type of their parameters on the fly.
The key is to insert this declaration at the right place, as it has to be within the same namespace as the original class and at the top of the source file. Otherwise there would be compile error as methods get called, which aren't already defined.
In this implementation the refactoring creates a new namespace area for the implementation class and the additional definitions of the original class to be inserted to, as inserts in the existing namespace are a problem for the underlying framework at the moment, if the insert points are also modified later on, which is highly likely.

4.7.6. deletePrivateLabels(), deletePrivateMembers()

Any existing private or protected members and their labels are no longer of use in the original class. They get removed.
We decided to remove the visibility labels here as well, to get a well defined state of the class after this operation, as C++ allows the programmer to freely mix private, protected and public members. This way, we can be sure no obsolete visibility labels stay in the code.

4.7.7. insertPointer()

The declaration of the pointer holding the reference to the implementation class gets inserted by this method.
As all private labels have been removed, a new label is inserted. The declaration is created using the parameters specified in the wizard.
This declaration also serves as a forward declaration for the implementation class.

4.7.8. changePublicMembers()

As the final step, this method transform all public members of the original class. At this point, those members are still untouched. The code body is dropped and different modifications need to be applied, depending on the role of the method.
Constructors get a memory initializer instantiating an implementation object and storing the reference to it into the implementation pointer.
Destructors receive a delete statement if the user has chosen a standard pointer. Otherwise the destructor is not needed and therefore removed.
Copy constructors instantiate the pointer to implementation with a copy of their implementation class.
The other "normal" members just delegate their call with all parameters to the implementation and eventually pass the return value to the caller.

4.7.9. Why this approach failed

The basic problem of this approach was, that you have to follow some limitations to program with the ASTRewrite. One important rule doesn't allow you to remove already inserted nodes. Each node which is once inserted by the ASTRewrite can't be eliminated at a later time during the refactoring process.
In our case the method collectModification(), as it's described above, iterated over all declarations and it's related definitions and created collections. The rest of the refactoring referred to this collections. The unresolveable problem which results is, that if the definitions were inline implemented in the header file, the collectMember() method saved the definition in the collection and directly copied it into the source file. The method additionally generated a declaration, which also was saved in the collection and replaced the definition in the header file. This means, that both nodes already were inserted by the ASTRewrite. In the later process it was impossible to remove such nodes. Specially copy constructors may have to be removed, or at least made private.
With this experience we created a new concept, which takes node by node and inserts them in the right state to the right place. So we are sure, that we never have to edit a node twice.

5. Perspective

Refactoring for C++ is a big field that needs a lot more work to be done. We just added one more part of the puzzle. Here we have some suggestions what to do in the future.

5.1. Improve the "Introduce PImpl" refactoring

During programming of this refactoring we had to make some sacrifice to the functionality. The handling of static methods could be improved so it detects calls to such methods and change them to qualified calls in the style of Class::staticMethod(). A part that we didn't checked at all is the handling of makros. Overall, there exists some limitations (limitations of the refactoring 4.5.7) because of not existing node handler.

5.2. Further tasks

This task amongst others have to be fulfilled to increase the abilities of CDT to refactore C++ code.

- To allow in future to develop new refactorings according to "Test Driven Design" (TDD) the refactoring test tool has to be improved. At the moment it's very strict in the way that the result of the test has to be exactly the same as the expected code, event all whitespaces must match. This is a problem since the code creation isn't that predictable. In fact we had to write first the refactoring and then adapt our test code so we could use it to test the refactoring work we did to our own code. But this approach is far away from TDD where one should write the test first and then implement the function accordingly.
- There are some bugs in the rewriter functionality of the AST (**??**) that we encountered. As an example to call an insert to insert node B before some node A first, and then use the rewriter again to replace that node A with a node C fails, because the rewriter does not know anymore what node A is where he should insert B.

Listing 5.1: Rewriter Problem

```
rewriter.insertBefore(nodeA, nodeB);
// some more code
rewriter.replace(nodeA, nodeC);
```

- Introduce more refactorings.There are a lot more mentioned in the Literature ([Fowler99]) that are worth to be implemented for CDT. If you just compare the number of refactorings for Java and C++ in eclipse, everyone sees that there is a lot of work to do.

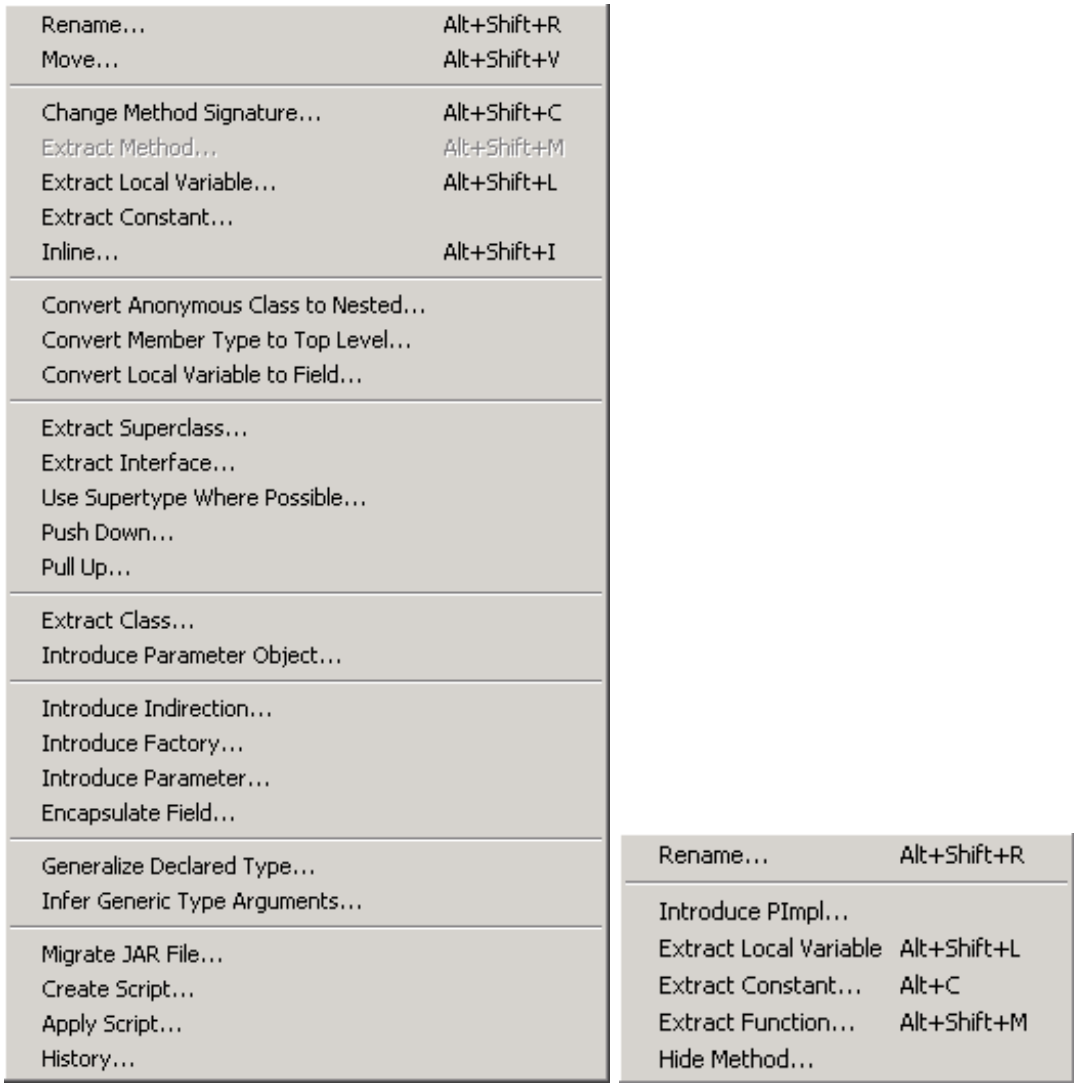


Figure 5.1.: Compare the number of refactorings in JDT and CDT

The most important ones are:

- Change method signature
- Move

And then there are some special refactorings just for C++

- Create implementation file from inline class
- Create Macro
- Inline Macro

- Design new handler for all possible node types in a class declaration to improve the limitations (4.5.7).

6. Result

In this work we succeeded in two main goals. First we showed that it's possible to make a complicated refactoring for C++ and second we fulfilled the wish of Alisdair Meredith a member of the C++ Standard committees [ACCU09].
In the end the most work was to get to know the AST and the rewriters and learn how to use them. The fact that we rewrote most of our code one week prior to the deadline because we found a problem with the rewriters shows that for creating new refactorings the actual code process isn't the problem but the complex framework and the lack of documentation is.

6.1. Introduce PImpl Idiom

We now have a funcitional refactoring for which we suggest to be integrated in the official CDT build after some more intensive testing and review. It covers most of the usual cases one could find in C++ code and returns working results for that code.

6.2. Refactoring Framework

We got to know the refactoring framework for C++ quite well. It's already pretty useful and allows the quick generation of at least simple new refactorings. with a complex refactoring like ours found a couple of bugs, some of them where just minor fixes like checking for null other on the other hand where bigger problems that demanded from us to rewrite a lot of code or abandon some features like creating a new file.

6.3. Eclipse Environment

With the help of some Literature [ClaybergRubel09] it was no problem to create the wizard. In fact most of the tasks Eclipse are already automated and just have to be customized. For example the compare view looks complicated but in fact is pre fabricated by the Eclipse framework.

6.4. Thanks

We had our best help resource in-house. Emanuel Graf was very helpful since we could contact him at almost at any time with questions about the refactoring framework and CDT. For help with the build process Lukas Felber was our contact. And then of course our advisor Prof. Peter Sommerlad was always a good help concerning questions about C++. We would like to thank all these people for their support of our semester thesis.

A. User guide

A.1. Download and install eclipse

A.1.1. Get eclipse

If you don't have the CD of the project, first you have to download an eclipse CDT release. The file is a compressed zip file, which only have to be unpacked to the local disk.
The official download of the CDT distribution[1].

A.1.2. Get the right CDT version

The actual CDT build (version 6.0.1) on the internet contains some failures in the refactoring framework. The introduce PImpl refactoring at least needs the version 6.0.2 of the cdt distribution. So if you downloaded an older version, you have to update the cdt, otherwise you can proceed with "Install the introduce PImpl refactoring plugin". To update the CDT follow the next steps.

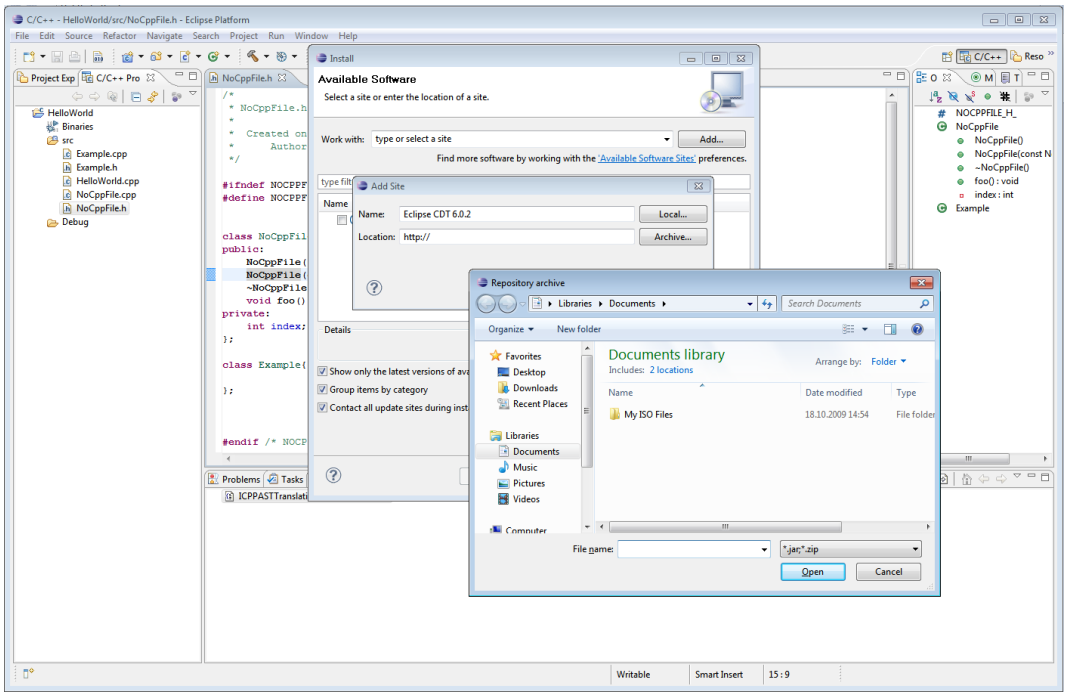


Figure A.1.: Install CDT update

[1] http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/galileo/SR1/eclipse-cpp-galileo-SR1-win32.zip.

1. If you don't have the CD of the project, you may find a download somewhere on the internet.
2. Open eclipse.
3. Goto Help>Install New Software...
4. Press the Add... button.
5. Insert any name.
6. Press the Archive... button and choose the downloaded CDT distribution.
7. Press OK.
8. Press the Next button and follow the instruction displayed on the screen.
9. After this process you have to restart eclipse.

A.2. Install the introduce PImpl refactoring plugin

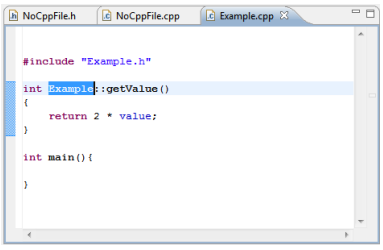
You can find the standalone plugin on the CD. To install the plugin, you have to open the folder containing the eclipse CDT distribution. The only thing you have to do there is to unpack the contents of the introduce_pimpl_refactoring_plugin.zip into the eclipse folder and override the dropins folder. So the plugin will be loaded on the startup of eclipse.

A.3. Start and use the plugin

If the plugin is installed correctly, you can find and run the plug in the refactoring menu. Note, that the plugin only works if you run it out of the following views:

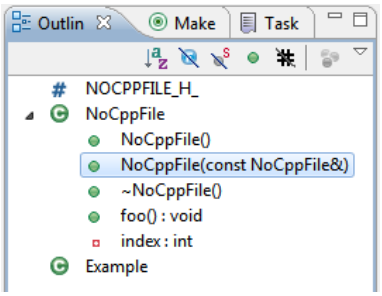
- Editor

  If a header file is opened in the editor, the selection doesn't matter. Otherwise a source file is opened in the editor, the selection have to be on a fully qualified function, that means, the function has to have a full nem with class declaration. So the refactoring is able to resolve the header file. Otherwise the refactoring displays selection invalid.



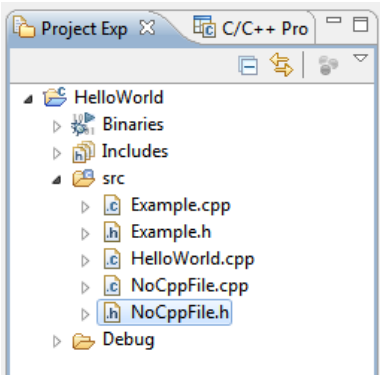
- Outline

  The selected node in the outline has hierarchically to be under a class node or a function definition with qualified name, otherwise it doesn't run.



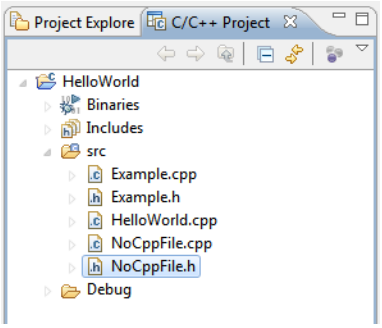
- Project Explorer

  Only header files are supported. If no header file is selected in the project explorer the refactoring doesn't run.



- C/C++ Projects

  Same as project explorer.



B. Management

B.1. Team

Andrea Berweger
Framework, Wizard
Implementing

Matthias Indermuehle
Server environement, LaTeX
Implementing

Roger Knoepfel
PImpl research, AST rebuilding
Implementing

B.2. Project plan

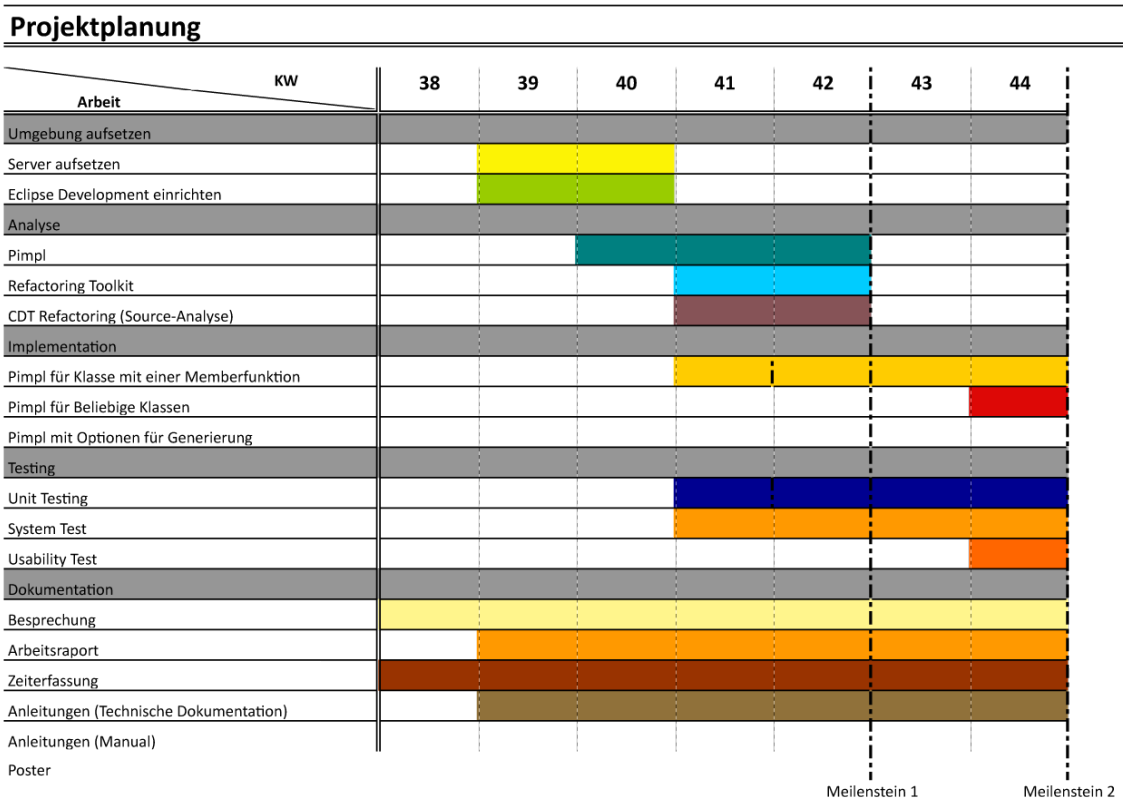


Figure B.1.: Work packages Part 1

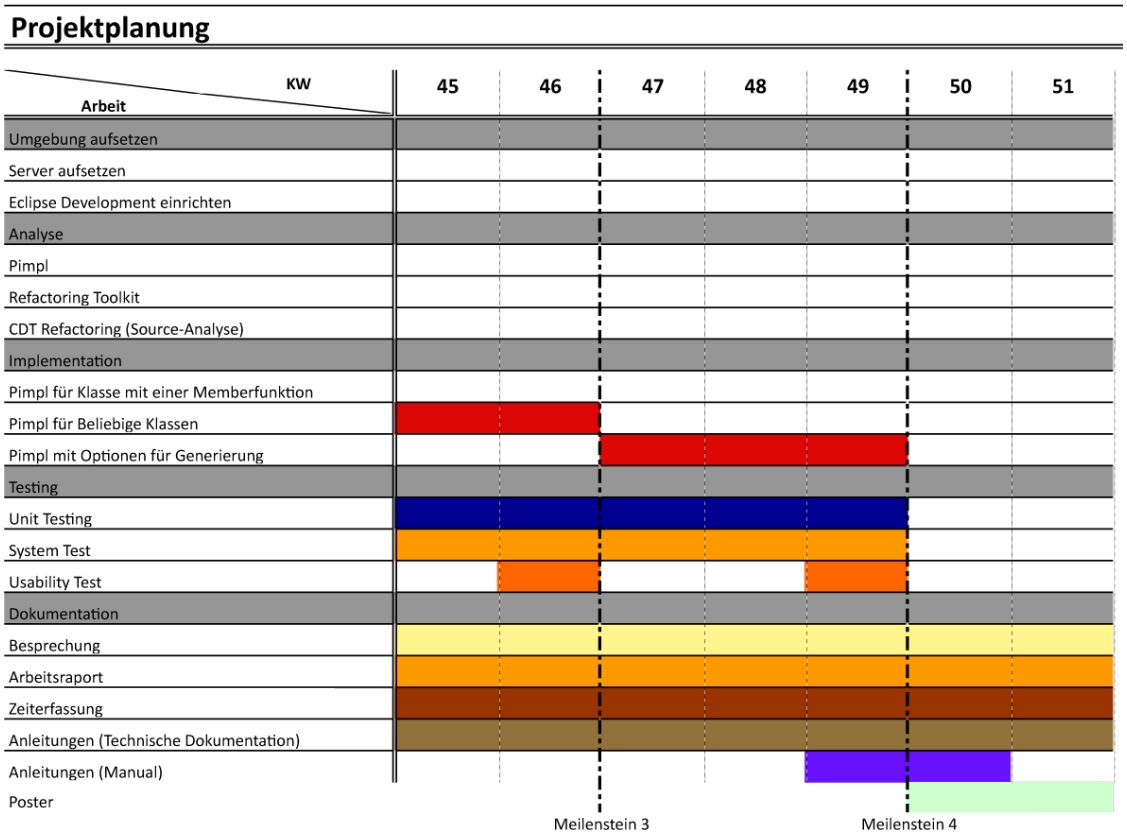


Figure B.2.: Work packages Part 2

Weekly meetings were scheduled at 14:00 every thursday.

B.3. Actual progress

We couldn't start to implement as planned as we had serious problems to build an Eclipse version with our plug-in loaded. This kept us off implementing for three weeks. We continued researching during this time.

We then started directly to implement a full version of the refactoring. Despite having further problems, mainly with unexpected behavior of the "ASTRewrite" class and the creation of correct nodes, we made good progress and were very close to implement all features we planned.
At the evening of friday 11th december Andrea discovered a serious problem with our entire concept of how to rebuild the code (4.7.9). The team met unscheduled the day after and decided to redo a major part of the code, implementing a new concept (4.5).
Fortunately, we managed to get the new concept running until sunday evening, leaving just minor bugfixing to do for the last week. We successfully asked to postpone the deadline though, to have enough time for a proper documentation.

B.4. Time spent

As with every project the time spent every week for the project increased over time. At the beginning we where a bit under the seventeen hours we should have worked on the thesis. but soon it flipped and we used much more time to code and write documentation. In week thirteen we discovered a big error (4.7.9) and had to redesign and re-implement a large part of our refactoring. The last week was occupied with mostly writing the documentation and fixing some minor last minute bugs.

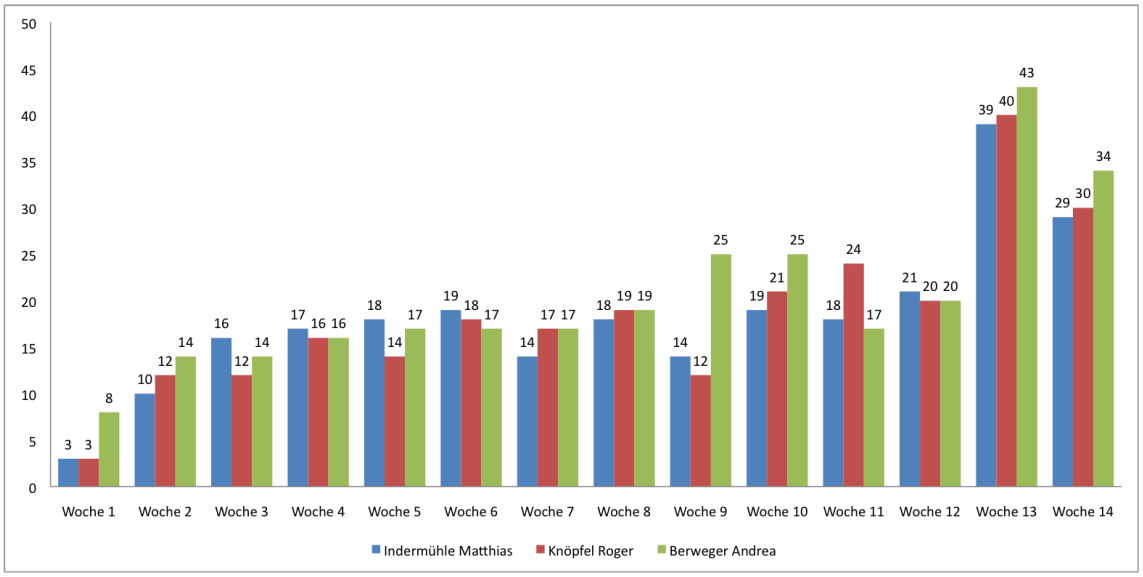


Figure B.3.: Timeload every week

As expected the implementation took away about half the time we had spent on this thesis. To our surprise the preparation of our environment took also a lot of time. This was caused by the fact that the automated build process for Eclipse Plugins is not trivial (C.3.2). The documentation was an ongoing task but climaxed as usual in such student projects in the last week. For the next work we should try to use even more the Bug/Feature feature of Redmine so we could take big parts of the documentation straight from there and the wiki.

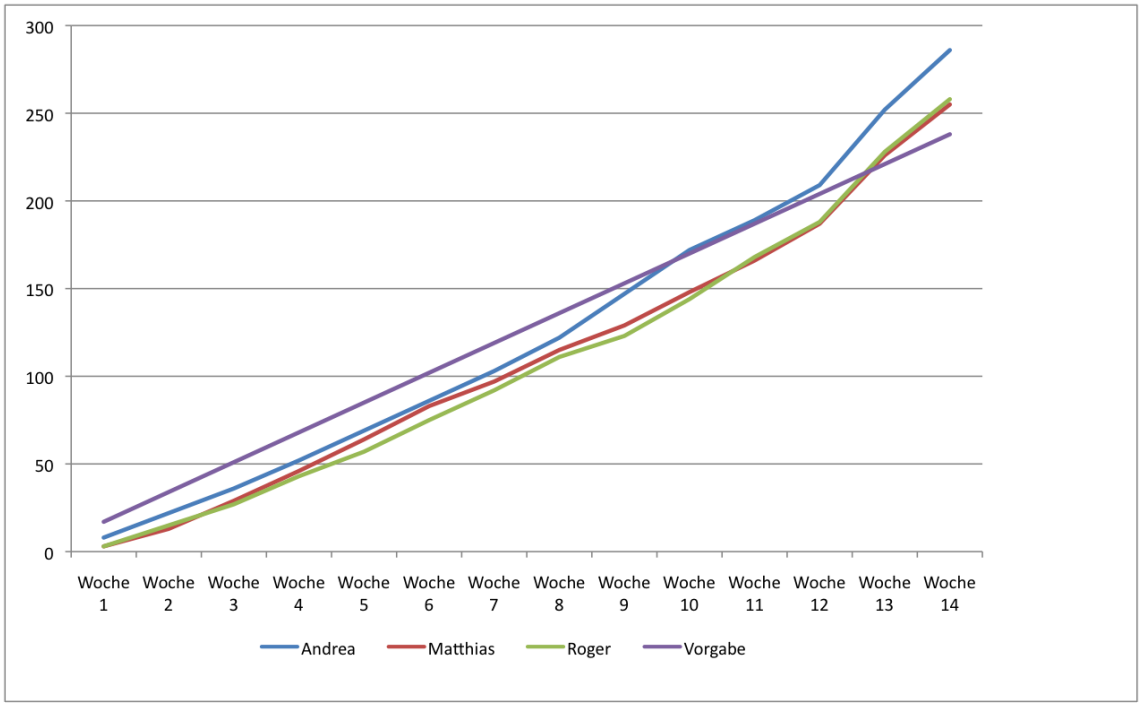


Figure B.4.: Working hours as sum over time

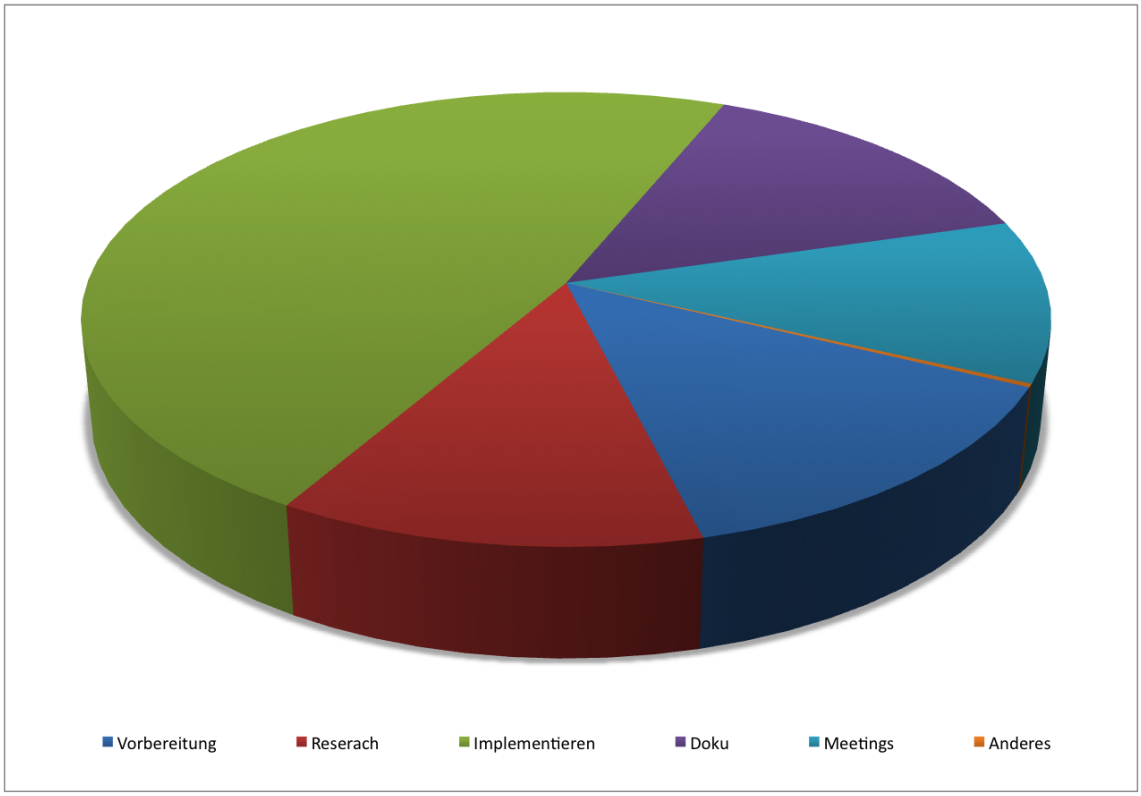


Figure B.5.: Working hours distributed over the working packages

B.5. Personal Review

B.5.1. Andrea Berweger

At the beginning, it was definitely not the project I wanted, but now I must agree, that it was a very interessting thesis. Indeed, I always wanted to learn the C++ syntax a little bit deeper, so I took this project as a chance. Now, I'm surely now what the PImpl idiome is, but I didn't learn much about the C++ syntax. Therefor I learned very much about refactoring, plugin development in java, that's surly a useful knowledge for the future, and work together with a very big existing project. First I tought I never will see through this big framework and faithfully I still don't, but I have learned that is not nessecary to overview to whole thing. You only need to understand the needed API and perhaps a short distance in. Surely, it was a little bit easier because the support of our supervisor was great. Additionally special thanks to our supervisors. The fact that our team is composed by three people, which no one has know the other, was never a problem. We ever could split our work and each man always knew what he has to do. There were never difficulties between the people. It was like a symbiosis. The project has applied, that theoretical each project have to be done twice. Because the first approace is to collect know how and problems, and the second one is to apply the learned content. It shows how tricky it is to create a perfect code in one chance. It's nearly impossible, but now we are (or should be) software engineers and we will give our best for future.

B.5.2. Matthias Indermühle

I planed to do a semester thesis with some friend of mine. But he ended up not having enough points to be allowed to write it. Instead I joined up with Andrea and we applied for some fancy GUI Thesis. After we got declined we choose this topic. After one week Roger joined our team. Even though we didn't know each other at first, but it worked out very good. At first I was sceptic about our tasks but soon I got fascinated by this big Framework. After some weeks, the first downer got to me. We didn't had written one single line of code, we just did a lot of research. That was not what I expected from a software engineering project.
After the sixth week the fun started, the first lines were written. From there on I learned a lot about how important the environment is. How nice it is to have a build server, a project management software and all these things. From the point of the code I learned two things.

- It would be best to do your project twice, once for getting comfortable with the framework and afterwards to do it right. Since that does not work so good, it's important to analyze the framework exactly so you don't run into one way streets.
- Sometimes it's just intelligent to trash a lot of code even though you are one week prior to the deadline. Don't try to safe all work, sometime it's more efficient to just start again.

B.5.3. Roger Knöpfel

Developing a C++ refactoring wasn't my first choice and i didn't know Matthias and Andrea before, but the project turned out to be very interesting and challenging. I never

worked within such a big framework and i think i learned much about adapting to existing concepts.
After a difficult start reading much about Eclipse plug-in development and finding my way through de CDT source code i was able to make good progress producing results in reasonable time at the end.
The crash of our first concept made clear how important it is to first learn to handle a framework before actually start to develop a new concept. I'm very glad we managed to rebuild everything in extremely short time and in the end had code that was not only working properly but also was much better structured, readable and also faster. Especially Andrea did an excellent job in this difficult moment.
I'm glad to have been part of this project. I think learned much about software development and look forward to create more refactorings with Matthias next year.

C. Project-server

Today software development does not happen anymore in a dark basement sitting alone nights after nights there with pizza and coffee. It has become a team effort with a need of communication. To provide the platform for working together was one of the first tasks we had to do. Our environment consist of three parts:

- a project management platform with a wiki and bug/feature tracking system[1] (Redmine C.1)
- a versioning system[2] (Svn C.2)
- an automated build server[3] (Hudson C.3)

Our project-server is a virtual machine provided by the IT team of the computer science department.

C.1. Project management platform - Redmine

Redmine is a open source project management platform written in Ruby on rails. Features that were important for us:

- User management
- A Wiki
- Document management
- Versioning integration
- Bug/Issue tracking system
- Feeds and E-mail notification

C.1.1. Installation

Eclipse runs on Ruby on rails so i had to install first these packages After some searching on the internet I found a nice How-To to setup redmine [Drinkingbird09]. The steps I had to do very most of them pretty obvious:

1. Install the ubuntu packages
2. Install the ruby gems

[1] http://en.wikipedia.org/wiki/Project_management
http://en.wikipedia.org/wiki/Bugtracker
[2] http://en.wikipedia.org/wiki/Revision_control
[3] http://en.wikipedia.org/wiki/Continuous_integration

3. Set up the database for Redmine use
4. Install Redmine itself
   a) Configure database access
   b) Bootstrap the database
   c) Setup mail notification for Redmine
5. Setup subversion connection
6. Configure apache with ModProxy
7. Set baseURL in 'config/environment.rb' [RedmineFAQ09]
8. Create startscript for redmine

C.1.1.1. Database connection

The database connection is defined in config/database.yml

Listing C.1: config/database.yml

```
production:
   adapter: postgresql
   database: redmine
   host: localhost
   username: redmine
   password: r3dm1n
   encoding: utf8
```

C.1.1.2. Apache Configuration

In apache the redmine is configured as a subside with a mod_proxy to the redmine ruby server.

```
   <Location /redmine/>
      Order allow,deny
      allow from all
      RewriteEngine On
      RewriteBase /redmine
      ProxyPass http://localhost:3000/
      ProxyPassReverse http://localhost:3000/
   </Location>
```

C.1.1.3. Startscript

With the redminestart.sh the Redmine instance could be easy restarted manually if you are logged in as root

Listing C.2: redminestart.sh script for restarting Redmine

```
#!/bin/bash
echo Start Redmine
cd /var/www/rails_apps/redmine-0.8/
mongrel_rails start --environment=production
```

C.1.2. Wiki

Redmine provides a Wiki plugin that we used for notes about our work and research. It supports to track back all changes and is great for writing down all thoughts and ideas. Unfortunately we didn't used it as much as we should have since then writing the documentation would be facilitated by the ability to recall all our thoughts from the Wiki.

C.1.3. Feature/Bug Tracker

Redmine has a build in Feature/Bug tracker where you can create a ticket if you find a bug or want to implement a new feature. It allows to specify a start and end Date, assign the task to a specific person, records time spent on the task, displays progress and hosts comments and notes about the problem.
We tried to use this feature, but where not entirely successful with it. We opened a ticket for almost every problem and bug but where a bit lazy in updating the tickets, write comments and close them after finishing.

C.1.4. Client

In addition to the webinterface for Redmine we used the Mylyn[4] task list for Eclipse. It allows to hide the parts of our project that we don't need for the current task and keep the IDE free from distraction.
For using Mylyn we had to install a connector every client and a mylyn plugin for the server [Mylyncon09].

C.2. Versioning - Subversion

Subversion (SVN) is a versioning system. It is used to store and maintain current and older version of the projects files, source code and documentation.

C.2.1. Installation

C.2.1.1. Server install and configuration

The installation of Subversion on Ubuntu 8.04 LTS is pretty straight forward since there are packages in the the repository. [UbuntuDoc09]

```
sudo apt-get install subversion libapache2-snv
svnadmin create /home/svn
```

[4] http://www.eclipse.org/mylyn/

For access over WebDAV provided by Apache2 one has to create a new subside in /etc/apache2/sites-aviable/default

```
<Location /svn>
  DAV svn
  SVNPath /home/svn
  AuthType Basic
  AuthName "Your repository name"
  AuthUserFile /etc/subversion/passwd
  <LimitExcept GET PROPFIND OPTIONS REPORT>
     Require valid-user
  </LimitExcept>
 </Location>
```

C.2.1.2. Client installation

As client we used Subversion[5] for the Eclipse platform.

C.3. Automated building system - Hudson

Hudson is a continuos integration system written in Java. it runs inside a container servlet server like Apache Tomcat. With continuos integration and testing errors and problems are discovered early in the process.

C.3.1. Installation

For Debian based Linux systems the installation is pretty straight forward. There is a repository with a package that can be installed with apt-get. [Hudson-ci09]

C.3.2. Build and Testing process

Building a plugin for Eclipse is different from building just any Java application because the process has a lot of dependencies to the "Eclipse Plugin Development Environment" (PDE). Building a Plugin for CDT is again one step higher. Not only the PDE is needed during the build process but also the CDT source.
For testing the plugin can not just be started on its own. There is a need for an environment consisting of an Eclipse with installed CDT plugin.

[5] http://subclipse.tigris.org/

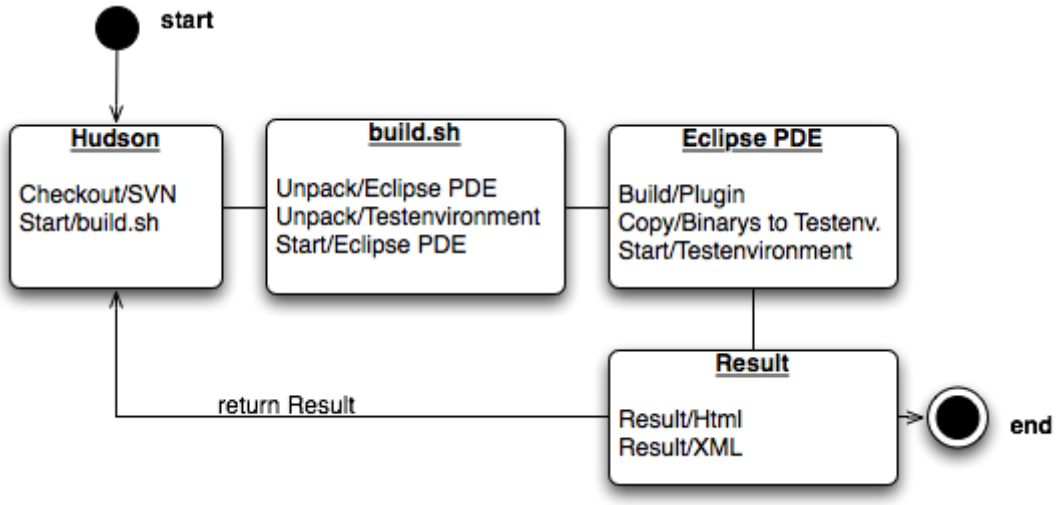


Figure C.1.: Buildprocess for the Plugin

C.3.2.1. Headless Build Process

For an automated build on a headless server some additional tools are needed. The Eclipse project provides a buildenvironment named Release Engineering (RelEng) with special ANT-Buildfiles and -Tasks. For executing these files the plugin org.eclpise.ant.core.basebuilder makes a whole Eclipse IDE without the GUI available. An alternative to that plugin is a parameter for the basic Eclipse to start it without head.

Listing C.3: Parameter to start Eclipse without head

```
-application org.eclipse.ant.core.antRunner
```

C.3.2.2. Problems

We had problems with testing our plugin automated because of two problems. Once we used a patched version of the CDT and not the official release version and on the other hand, we used a testframework written at the HSR that did not work together with the automated testing.

D. AST node descriptions

This section is copied and translated of the research project "C++ Refacoring Support for Eclipse CDT" created by Graf Emnauel and Leo Büttiker.[GrafButtiker06]
This section explains the different nodes of the CDT model used in a AST. The most nodes are expanded by a code example. The design of the nodes is oriented by the C++ standard.

D.1. IASTTranslationUnit

IASTTranslationUnit

The IASTTranslationUnit represents the root node of a file, respectivly the root of an Abstract Syntax Tree (AST 3.2.1). That means the this node represents the whole file, and all other nodes are appended to this node.

D.2. IASTDeclaration

IASTDeclaration

Basis-Interface of all declarations.

IASTASMDeclaration

An inline assembler declaration of the C++ standard. To compare two nodes you have to compare the string with the assembler declaration.
Structure
asm ("string-literal");
Example

```
asm ("movl %1, %%eax;");
```

IASTFunctionDefinition

The definiton of a function.
Structure
IASTDeclSpecifier IASTFunctionDeclarator IASTStatement
Example

```
HalloWorld::hello(int *const lp,int i,...)
{
i++;
```

```
}
```

IASTProblemDeclaration

A problem in a declaration.
Structure
throws ProblemRuntimeException

IASTSimpleDeclaration

A simple declaration.
Structure
IASTDeclSpecifier IASTDeclarator*
Example

```
int *const lp, *const volatile gp, *const volatile restrict ip;
```

ICPPASTExplicitTemplateInstantiation

Explicit instantiate of a template (de-c.13.9).
Structure
IGPPASTExplicitTemplateInstantiation$_{\text{opt}}$template IASTDeclaration
Example

```
template class vector<int>;
```

IGPPASTExplicitTemplateInstantiation

Support for three expansion declarations of the GNU C++ compiler. The expansion declarations have to be compared by getModifier(). http://gcc.gnu.org/onlinedocs/gcc/Template-Instantiation.html#Template-Instantiationweb
Structure
(static |inline |extern) template IASTDeclaration
Example

```
static template class vector<int>;
```

ICPPASTLinkageSpecification

Defines the linkage language for non C++ fragments. The string literal have to be compared by a node comparision.
Structure

extern string-literal IASTDeclaration* |extern string-literal IASTDeclaration
Example

```
extern "C" typedef void FUNC();
```

ICPPASTNamespaceAlias

Definition of a namespace alias.
Structure
namespace IASTName = IASTName;
Example

```
namespace Kurz = ZiemlichLangerNamespaceName;
```

ICPPASTNamespaceDefinition

Definiton of a namespace.
Structure
namespace IASTName IASTDeclaration*
Example

```
namespace ns
{
int i;
double j;
}
```

ICPPASTTemplateDeclaration

A template declaration. The flag "isExported" have to be compared.
Structure
export $_{opt}$template<ICPPASTTemplateParameter*>
Example

```
template <typename T>
class MyQueue<T,T>{};
```

ICPPASTTemplateSpecialization

A template serialisation.
Structure
template <> IASTDeclaration
Example

```
template <> void MyQueue<double> ::Add(double const \&d)
{
data.push_back(d);
}
```

ICPPASTUsingDeclaration

Declarates a name of a field or method of a super class in the own class.
Structure
using typnameopt IASTName;
Example

```
using typename A::k;
```

ICPPASTUsingDirective

Imports a namespace. The flag "isTypename" have to be compared.
Structure
using namespace IASTName;
Example

```
using namespace B;
```

ICPPASTVisiblityLabel

Visibility definition in a class declaration. Compare of the nodes with aid of getVisibility.
Structure
private: |public: |protected:
Example

```
private:
```

D.3. IASTDeclSpecifier

IASTDeclSpecifier

Basic interface for all declaration specifiers. Defines different specifier types: typedef, extern, static, auto, register. To compare use the storage class and the flags "isConst", "isInline" and "isVolatile".

ICASTDeclSpecifier

C extension of the basic interface. C allows the additional key word restrict. To compare use additionally the flas "isRestricted".
Structure
restrict ICASTDeclSpecifier

ICPPASTDeclSpecifier

CPP extensio of the basic interface. C++ allows the key words explicit, friend and virtual. It also defines an additional specifier type mutable. To compare use additionally the flags "isExplicit", "isFriend" and "isVirtual".

IGPPASTDeclSpecifier

Extends DeclSpecifier with the GNU specified extension "isRestrict".

IASTCompositeTypeSpecifier

Represents a composite structur, which can contain more declarations. Defines the types struct and union. To compare use the getKey() method.
Structure
(union |struct) IASTName IASTDeclaration* (Members)
Example

```
struct s {int a;};
```

ICASTCompositeTypeSpecifier

Extends ICASTDeclSpecifier and IASTCompsiteTypeSpecifier. Defines no new methods or constants.

ICPPASTCompositeTypeSpecifier

Extends ICPPASTDeclSpecifier and IASTCompsiteTypeSpecifier. Defines additionally the type "class".
Structure
class IASTName(: BaseSpecifiers)$_{\text{opt}}$ IASTDeclaration*
Example

```
class TestClass2 : public TestClass{}
```

IASTElaboratedTypeSpecifier

An elaborated type specifier is a name after a key word enum, struct, union, or class. The elaborated type specifier is used to instantiate an object of a predefined declarator type. The name identifies the type of the ew oblect. Example: enum Status good,

bad;/*Typ Deklaration*/ enum Status stat;/*Elaborated Form*/ Status stat2;/*not elaborated form*/ The interface defines constants for key words enum, struct and union. To compare use these constants with aid of the method ”getKey()”.
Structure
type-keyword IASTName
Example

```
enum Status stat;
```

ICASTElaboratedTypeSpecifier

Extends ICASTDeclSpecifier and IASTElaboratedTypeSpecifier. Defines no new methods or constants.

ICPPASTElaboratedTypeSpecifier

Extends ICASTDeclSpecifier and IASTElaboratedTypeSpecifier. Defines additionally a constant for the type ”class”.
Structure
class A * A;

IASTEnumerationSpecifier

Spezifiziert eine Enumerationsdeklaration.
Structure
enum IASTName IASTEnumerator*
Example

```
enum MyEnum { A = 10, B = 20}
```

ICASTEnumerationSpecifier

Extends ICASTDeclSpecifier and IASTEnumerationSpecifier. Defines no new methods or constants.

IASTNamedTypeSpecifier

Represents the usage of a typedef name.

ICASTTypedefNameSpecifier

Extends ICASTDeclSpecifier and IASTNamedTypeSpecifier. Defines no new methods or constants.

ICPPASTNamedTypeSpecifier

Extends ICPPASTDeclSpecifier and IASTNamedTypeSpecifier. Defines a flag for typname. To compare use additionally the flags "isTypname", "isExplicit", "isFriend" and "isVirtual".

IASTSimpleDeclSpecifier

The specification of a fundamental type. Defines flags for the qualifiers signed/unsigned and long/short. Also defines constants for these types. To compare use additionally the flags "isLong", "isShort", "isSigned" and "isUnsigned".
Structure
$(\text{signed}|\text{unsigned})_{\text{opt}}$ $(\text{long}|\text{short})_{\text{opt}}$ type-name
Example

```
signed long int i;
```

ICASTSimpleDeclSpecifier

Extends ICASTDeclSpecifier and IASTSimpleDeclSpecifier. Defines new methods for C constants. To compare use additionally the flags "isRestrict", "isComplex", "isImaginary" and "isLongLong".

ICPPASTSimpleDeclSpecifier

Extends ICPPASTDeclSpecifier and IASTSimpleDeclSpecifier. Defines constants for C++ types bool and wchar_t.

IGPPASTSimpleDeclSpecifier

Extends ICPPASTSimpleDeclSpcifier with GNU specific components. It's possible to set a TypeofExpression. In addition the flags "isComplex", "isImaginary" and "isLongLong" will be added.

D.4. IASTDeclarator

IASTDeclarator

A declarator is a part of a declaration.
Structure
$\text{IASTPointerOperator}_{\text{opt}}$ IASTName $\text{IASTInitializer}_{\text{opt}}$

IASTArrayDeclarator

Extends IASTDeclarator with the ArrayModifier.
Structure
$\text{IASTPointerOperator}_{\text{opt}}$ IASTName IASTArrayModifier $\text{IASTInitializer}_{\text{opt}}$
Example

```
int[3]
```

IASTFieldDeclarator

The declaration of a bit field.
Structure
IASTPointerOperator$_{opt}$ IASTName : IASTExpression IASTInitializer $_{opt}$
Example

```
int bf : 10
```

IASTFunctionDeclarator

The declaration of a function.
Structure
IASTPointerOperator$_{opt}$ IASTName (IASTParameterDeclaration*)

IASTStandardFunctionDeclarator

The declaration of a function. It have to be checked with aid of the method "takesVarArgs()" if the function takes variable arguments.

ICPPASTFunctionDeclarator

The declaration of a C++ function. It also can be a constructor. To compare is recommended to use the flags "isConst", "isPureVirtual" und "isVolatile".

ICPPASTFunctionTryBlockDeclarator

The declaration of function with a try block.

ICASTKnRFunctionDeclarator

The declaration of a function in K&R notation.
Structure
IASTName ((IASTName ,)* IASTName) IASTParameterDeclaration
Example

```
int foo(a,b)
int a, b;
```

D.5. IASTStatement

IASTBreakStatement

The termination of a loop.
Structure
break;
Example

```
break;
```

IASTCaseStatement

The case statement in a switch case syntax.
Structure
case IASTExpression:
Example

```
case '1':
```

IASTCompoundStatement

Multiple statements, in most cases the content of a loop, method or a branch. Line break after closing brackets have to be avoided.
Structure
IASTStatement*
Example

```
{
doOne();
doTwo();
}
```

IASTContinueStatement

The continuation of a loop. Jumps to the end of a loop and continues it.
Structure
continue;
Example

```
continue;
```

IASTDeclarationStatement

IASTDeclaration

IASTDefaultStatement

A default statement of a switch case syntax.
Structure
default:
Example

```
default:
```

IASTDoStatement

A do while loop.
Structure
do IASTStatement while(IASTExpression);
Example

```
do{
something();
}while(true);
```

IASTExpressionStatement

A simple Expression.
Structure
IASTExpression;
Example

```
i++;
```

IASTForStatement

A for loop.
Structure
for(IASTStatement;$\text{IASTExpression}_{\text{opt}}$;$\text{IASTExpression}_{\text{opt}}$)writeBodyStatement(IASTStatement
)
Example

```
for(int d=0;d<10;++d) {
d++;
}
```

ICPPASTForStatement

A C++ for loop.
Structure
for(IASTStatement;IASTExpression$_{opt}$|IASTDeclaration$_{opt}$;IASTExpression$_{opt}$)writeBody-Statement(IASTStatement)
Example

```
for(int d=0;d<10;++d) {
d++;
}
```

IASTGotoStatement

A goto statement.
Structure
goto IASTName;
Example

```
goto start;
```

IASTIfStatement

A C if-then-else statement.
Structure
if(IASTExpression)IASTStatement(True-Body)(else IASTStatement(False-Body))$_{opt}$
Example

```
if(i<0){
test();
}else{
test2();
}
```

ICPPASTIfStatement

A C++ if-then-else statement. C++ allows declarations in the condition.
Structure
if(IASTExpression|IASTDeclaration)writeBodyStatement(IASTStatement)(else writeBody-Statement(IASTStatement))$_{opt}$
Example

```
if(A a = i){
test();
}else{
```

```
test2();
}
```

IASTLabelStatement

A goto label statement.
Structure
IASTName:IASTCommments
Example

```
start:
```

IASTNullStatement

A null statement.
Structure
;
Example

```
;
```

IASTProblemStatement

A parser problem in a statement.

IASTReturnStatement

A return statement of a method or function.
Structure
return IASTExpression;
Example

```
return i*i++;
```

IASTSwitchStatement

A switch statement.
Structure
switch(IASTExpression)IASTStatement(Switch-Body)
Example

```
switch(){
case 'a':
```

```
do();
break;
default:
hallo();
}
```

ICPPASTSwitchStatement

A C++ switch statement. C++ allows declarations in the condition.
Structure
switch(IASTExpression|IASTDeclaration)IASTStatement(Switch-Body)

IASTWhileStatement

A C while loop statement.
Structure
while(IASTExpression)(IASTStatement(Schleifen-Body)
Example

```
while(i<0){
i++;
}
```

ICPPASTWhileStatement

A C++ while loop statement. C++ allows declarations in the condition.
Structure
while(IASTExpression|IASTDeclaration)writeBodyStatement(IASTStatement)
Example

```
while(A a = i){
i = 0;
}
```

ICPPASTTryStatement

A try block statement.
Structure
try IASTStatement
Example

```
try{
throw A;
}
```

ICPPASTCatchHandler

A catch block statement. It's possible to declare if all exceptions will be caught. This will be set with the flag "isCatchAll". To compare use this flag.
Structure
catch(IASTDeclaration|...) IASTStatement
Example

```
catch(...){
a++;
}
```

D.6. IASTExpression

IASTArraySubcriptExpression

An access to an array.
Structure
ASTExpression[ASTExpression]
Example

```
a[1]
```

IASTBinaryExpression

Binary expression. The operator will be defined through int constants. Oparator have to be compared with aid of the method "getOperator()".
Structure
IASTExpression Operator IASTExpression
Example

```
a + 1
```

ICPPASTBinaryExpression

Defines additional C++ operators.

IGPPASTBinaryExpression

Defines additional G++ operators.

IASTCastExpression

Represents a type cast. The type of this cast is defined by int constants. All implementing classes also implements IASTUnaryExpression. The comparison proceeds in IASTUnary-

Expression.
Structure
(IASTTypeId) IASTExpression (Expression which will be casted)
Example

```
(int) 'A'
```

ICPPASTCastExpression

Defines a C++ cast types.
Structure
dynamic_cast< IASTTypeID> IASTExpression static_cast<IASTTypeID> IASTExpression reinterpret_cast<IASTTypeID> IASTExpression const_cast<IASTTypeID> IASTExpression
Example

```
dynamic_cast<TestClass*>(tc);
```

IASTConditionalExpression

Triple expression with question mark.
Structure
IASTExpression (LogicalCondition) ? IASTExpression (PositveResult) : IASTExpression (NegativResult)
Example

```
c > i ? a = i : a = c
```

IASTExpressionList

A expression list, seperated by commas.
Structure
IASTExpression (Liste), IASTExpression
Example

```
s = 3, c = 4, h = 5
```

IASTFieldReference

An access to a field. It's possible to declare if it's a pointer dereference. To compare attend to this.
Structure

IASTExpression (FieldOwner)(.|->)IASTName (FieldName)
Example

```
a.b oder a()->def
```

ICPPASTFieldReference

A C++ access to a field can contain additionally the key word temlate. To compare attend to this template key word.
Structure
IASTExpression (FieldOwner)(.|->)$\text{template}_{\text{opt}}$ IASTName (FieldName)
Example

```
m.template get_new<int>()
```

IASTFunctionCallExpression

A function call expression.
Structure
IASTExpression (Functionname)(IASTExpression (Parameters))
Example

```
ausgeben(1)
```

IASTIdExpression

Represents an identifer.
Structure
IASTName
Example

```
z1
```

IASTLiteralExpression

Represents a literal. The type of this literal will be defined by int constants. The method "toString()" returns the value of the literal as String. If it's a character literal the string also contains the symbols ', otherwise if it's a string literal the string not contains the symbols ". To compare use first the the type with "getType" and after that compare the strings.
Structure
'A'

ICPPASTLiteralExpression

Defines additionally C++ literal types like "this" or "true".
Structure
true

IASTProblemExpression

Represents a parse problem in a expression.

IASTUnaryExpression

Represents an unary expression. To compare use the operator with aid of the method "getOperator()".
Structure
$\text{Prefixoperator}_{\text{opt}}$IASTExpression (Operand) $\text{Postfixoperator}_{\text{opt}}$
Example

```
++i
```

ICPPASTUnaryExpression

Defines the additional unary C++ operations "throw" and "typeid".

IGNUASTUnaryExpression

Defines the additional unary GNU operations "typeof" and "__alignOf"

IASTTypeIdExpression

The sizeof operator is declared by a int constant. This value is accessible through "getTypeId()". This way also the sub classes can by proofed.
Structure
sizeof (IASTTypeId)
Example

```
i = sizeof (int);
```

ICPPASTTypeIdExpression

Extends IASTTypeIdExpression with the C++ operator "typeid".
Structure
typeid (IASTTypeId)
Example

```
typeid (int);
```

IGNUASTTypeIdExpression

Extends IASTTypeIdExpression with the GNU operators "typeof" and "__alignof". These extensions are obviously not implemented in the parser. There are no classes which implements this interface.
Structure
typeof (IASTTypeId) __alignof(IASTTypeId)

ICPPASTDeleteExpression

The delete expression, which delets with "new" created objects. To compare use the flags "isGlobal" and "isVectored".
Structure
$::_{opt}$ delete IASTExpression $|::_{opt}$ delete [ ] IASTExpression
Example

```
delete n
```

ICPPASTNewExpression

Allocates new objects. To compare use the flags "isGlobal" and "isVectored".
Structure
$::_{opt}$new IASTExpression (Placement)$_{opt}$ IASTTypeId (IASTExpression (Initializer))
Example

```
new TestClass(b)
```

ICPPASTSimpleTypeConstructorExpression

Constuctor call with a fundamental type. To compare two fundamental types use the method "getSimpleType".
Structure
TypeId (IASTExpression)
Example

```
int(3)
```

ICPPASTTypenameExpression

The usability of this node is vague. Names normally will be declarated as IASTName. To compare use the template key word.

D.7. IASTName

IASTName

Represents a name.

ICPPASTQualifiedName

Extends IASTName and represents a qualified name.
Structure
IASTName::IASTName
Example

```
TestClass::getA
```

IASTConversionName

Extends IASTName and represents the name of a convert operator.
Structure
operator IASTTypeId
Example

```
operator int
```

ICPPASTOperatorName

Extends IASTName and represents the name of the operator function.
Structure
opeator Operator
Example

```
operator +
```

ICPPASTTemplateId

Extends IASTName and represents the name of a template instantiation.
Structure
IASTName (Template Name) < IASTNode (Template Parameter) >
Example

```
String<char>
```

D.8. IASTInitializer

IASTInitializerExpression

A simple initialization expression.
Structure
IASTExpression
Example

```
int(1) oder 3
```

IASTInitializerList

A list of initializations seperated by commas.
Structure
IASTInitializerList, IASTInitializer
Example

```
{1, {2,3}}
```

ICPPASTConstructorInitializer

A call of a constructor without the key word "new".
Structure
(IASTExpression)
Example

```
(20)
```

ICASTDesignatedInitializer

A Initialization with named parameter (only in C).
Structure
ICASTDesignator? = IASTInitializer
Example

```
.b.i = 2
```

D.9. IASTPointer

IASTPointerOperator

A pointer operator.
Structure

ICPPASTReferenceOperator |IASTPointer

ICPPASTReferenceOperator

A reference operator.
Structure
&

IASTPointer

A pointer. To compare use the flags "isConst" and "isVolatile".
Structure
(ICPPASTPointerToMember$_{opt}$ * const$_{opt}$ volatile$_{opt}$ ICASTPointer$_{opt}$) |(ICPPASTPointer-ToMember$_{opt}$ * const$_{opt}$ volatile$_{opt}$ ICPPASTPointer$_{opt}$)
Example

```
* const volatile
```

ICPPASTPointerToMember

A pointer to member. http://www.parashift.com/c++-faq-lite/pointers-to-members.htmlweb
Structure
IASTName
Example

```
A::*
```

IGPPASTPointer

The restrict key word for C++ extensions. To compare use additionally the flag "isRestrict". http://www.thescripts.com/forum/thread507409.htmlweb
Structure
* const volatile restricted

ICASTPointer

The restrict key word for C extensions. To compare use additionally the flag "isRestrict". http://developers.sun.com/sunstudio/articles/cc_restrict.htmlweb
Structure
* const volatile restricted

Bibliography

[Fowler99] Refactoring: Improving the Design of Existing Code, M. Fowler, Addison-Wesley, 1999

[Fowler05] Refactorin oder: wie Sie das Desing vorhandener Software verbessern, M. Fowler Addison-Wesley, 2005

[WikiPImpl09] Wikipedia, Opaque pointer
Website: http://en.wikipedia.org/wiki/Opaque_pointer, 2009-12-16

[Coplien92] Advanced C++, Programming Styles and Idioms, James O. Coplien, AT&T Bell Telephone Laboratories, 1992

[GrafButtiker06] Diplomarbeit: C++ Refactoring Support fuer Eclipse CDT, Emanuel Graf, Leo Buettiker, HSR Rapperswil, 2006

[ACCU09] Personal communication between Prof. Peter Sommerlad and Alisdair Meredith, member of the C++ Standard committees at the ACCU conference in Oxford, 200?

[ClaybergRubel09] Eclipse Plug-ins, Third Edition, Eric Clayberg, Dan Rubel, Addison-Wesley, 2009

[Drinkingbird09] Setting up a Redmine site on Ubuntu: Drinkingbird
Website: http://drinkingbird.net/blog/articles/2008/02/27/setting-up-a-redmine-site-on-ubuntu, 2009-10-01

[RedmineFAQ09] Redmine - HowTo Install Redmine in a sub-URI,
Website: http://www.redmine.org/wiki/1/HowTo_Install_Redmine_in_a_sub-URI, 2009-10-01

[Mylyncon09] Sourceforge.net:Installation - redmin-mylyncon,
Website: http://sourceforge.net/apps/mediawiki/redmin-mylyncon/index.php?title=Installation, 2009-10-27

[UbuntuDoc09] Subversion - Ubuntu official documentation
Website: https://help.ubuntu.com/8.04/serverguide/C/subversion.html, 2009-10-01

[Hudson-ci09] Installing Hudson on Ubuntu
Website: http://wiki.hudson-ci.org/display/HUDSON/Installing+Hudson+on+Ubuntu, 2009-12-09
Website: http://wiki.hudson-ci.org/display/HUDSON/Running+Hudson+behind+Apache, 2009-12-09

Glossary

AST An **Abstract Syntax Tree**, An abstract view of the source code. 2, 16

boost The **Boost C++ Libraries** are a set of open source libraries that extend the functionality of C++ . 8

CDT **C/C++ Development Tooling** is an Integrated Development Environment for Eclipse for C and C++, http://www.eclipse.org/cdt. 2

IDE An **Integrated Development Environment** is a software to write programs that provides in addition to the editor the whole toolchaint to build and debug the code. 2

JDT **Java Development Tools** is an Integrated Development Environment for Eclipse for Java, http://www.eclipse.org/jdt. 2

JUnit A unit test framework for Java http://www.junit.org . 15

TR1 The **TR1** is a Library Extention to the standart library of C++ . 8

Keyword Index