HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

INSTITUTE
FOR
SOFTWARE

HSR – University of Applied Sciences Rapperswil

Departement of Computer Science - Institute for Software

CDT C++ Refactorings

Bachelor Thesis: Spring Term 2010

Matthias Indermühle
mindermu@hsr.ch

Roger Knöpfel
rknoepfe@hsr.ch

Advisor:	Prof. Peter Sommerlad
Expert:	Martin Botzler
Co-Examiner:	Prof. Dr. Peter Heinzmann

Version: June 18, 2010

Abstract

Refactoring existing code is a key element in almost all modern software development. Typically, this is performed using tools of an integrated development environment, since manual changes are slow and error prone. The "C/C++ Development Tooling" plug-in for Eclipse (CDT) lacks some important refactoring tools. We have already developed the "Introduce PImpl" refactoring in our semester thesis at the IFS in 2009 and our goal for this bachelor thesis is to further expand the refactoring abilities of CDT.
Most C++ functions provide one or more declarations in separate (Header) files in addition to their definition. If the function signature must be changed the programmer has to edit code in these files and also the calling sites accordingly. This is annoying and increases the risk for mistakes. With our refactorings, the software developer specifies a single change that is properly applied to all the relevant parts of code.
To get an easier start with the Eclipse framework we began with a rather simple refactoring called "Declare Function" that offers the possibility to create a function declaration within a header file based on the function's definition. It is the counterpart to the already existing "Implement Method" (sic) refactoring.
Our main feature "Change Function Signature" allows changing the signature of a function with a very simple and yet powerful wizard. The user can add, remove or rearrange parameters, modify the return type and choose to apply those changes to its super- or subclass, instead of manually rewriting the definition, the declaration and all callers.
Both our refactorings combined with "Implement Method" frees C++ developers from the burden of having to change code in multiple files, when extending or changing a class' interface.

Contents

1. Motivation

"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]

Modern software development is performed in iterations, adding one feature after the other, continuously adapting and expanding existing code. This modification of working code by preserving its functionality is a key process in the daily work of every programmer. To ease this rather generic work, integrated development environments offer various automated refactoring tools. This allows the programmer to spend more time designing good software, rather than performing numerous simple changes on various points of a project.

1.1. Refactoring in C++

Programming in C++ follows many principles that increase the complexity of creating or changing code.

Because the C++ compiler traverses each file only one time, it is necessary to declare functions before they they are used. While in simple situations it is sufficient to change the sequence of function definitions in the code (as a definition is also a declaration), most of the time the solution are forward declarations.

Listing 1.1: Function definition before caller

```
int foo() {
   return 42;
}

int bar(){
   return foo();
}
```

Listing 1.2: Forward declarations

```
int foo(int); // Forward declaration of foo()
int bar(int); // Forward declaration of bar()

int foo(int i) {
   if (i == 0) return 1
   return bar(i) - 1;
}

int bar(int j){
   if (i == 1) return 1
   return foo(i-1) / 2;
}
```

Classes are normally written in two separate files: A header file holds the class declaration for clients to be included, while its function definitions are in an implementation file.

Listing 1.3: Header file - Example.h

```
class Example {
public:
   int foo(int);
private:
   int number = 2;
   int bar(int);
}
```

Listing 1.4: Implementation file - Example.cpp

```
#include "Example.h"
int  Example foo(int i) {
   if (i == 0) return 1
   return foo(i) - number;
}

int bar(int j){
   if (i == 1) return 1
   return foo(i-1) / number;
}
```

The "One Defintion Rule" [ISO14882] states, that a function can have any number of declarations.

Those concepts force a programmer to modify code at various places in order to perform a single logical change and therefore not only increase the effort for the programmer, but also the chance of errors. For this reason, automated refactoring tools are very valuable as the reduce both developing time and the number errors.

1.2. Refactorings in CDT

The Eclipse C/C++ Development Tooling (CDT) already offers a decent set of automated refactorings. However, in relation to the impressive amount of tools offered in Java Development Tools (JDT) CDT still lacks a lot of important features. Efforts are being made to supplement those tools and we would like to contribute our part to this work.

Table 1.1.: Comparison of available refactoring: JDT on the left, CDT on the right

JDT	CDT
Rename	Rename
Move	Extract Constant
Change Method Signature	Extract Local Variable
Inline	Extract Function
Extract Interface	Hide Method
Extract Superclass	Implement Method
Use Supertype Where Possible	
Pull Up	
Pull Down	
Extract Class	
Introduce Parameter Object	
Introduce Indirection	
Generalize Declared Type	
Infer Generic Type Arguments	

1.3. Introduce PImpl

In our preceeding work "Introduce PImpl Refactoring" [PImpl09] we already implemented a refactoring tool for CDT together with Andrea Berweger, learning much about the Eclipse framework in general and the development of plug-ins and refactorings in specific. As this thesis turned out to be a very interesting and motivating challenge, we decided to continue developing refactoring tools for CDT as our bachelor thesis.

1.4. Goal

A great amount of team expertise was lost, when Andrea Berweger, who wrote almost the entire GUI of "Introduce PImpl Refactoring" [PImpl09] and did a great job hooking the plug-in into Eclipse, was offered a different thesis at his work place. We decided to start by creating a rather simple refactoring, called "Declare Function" (3) to learn about those parts and also refresh our knowledge of developing C++ refactoring plug-ins. We then moved on to our main feature, the "Change Function Signature" (4) refactoring, which has to be highly configurable and whose execution can have a huge impact on many projects and files. Not only the definition, the declarations and the callers of the refactored function need to altered, but also possible parent and child classes.

2. Environment

Fortunately, we do not have to build our refactorings from scratch, as there is already a powerful framework available to change code and interact with the user. In this chapter, we will highlight the structure and explain what has to be done to hook up our plug-in to Eclipse and to make use of the offered tools and resources. We further reused some code from our precessor thesis [PImpl09].

2.1. The Language Toolkit

”Anyone who supports a programming language in an Eclipse-based IDE will be asked sooner or later to offer automated refactorings - similar to what is provided by the Java Development Tools (JDT). Since the release of Eclipse 3.1, at least part of this task - which is by no means simple - is supported by a language neutral API: the Language Toolkit (LTK)” [Frenzel05]

Originally, Eclipse was designed as a Java IDE and without any plug-ins it still is. To support the development of tools for other programming languages the Language Toolkit (LTK) was introduced. This framework offers support for operations and concepts common to all, or at least most of all, programming languages. Refactoring is one of those elements. This language neutral structure was extended to specifically support operations on C++ by de CDT developers. The following call diagram offers an overview of the classes typically involved inrefactoring process and of how they interact.

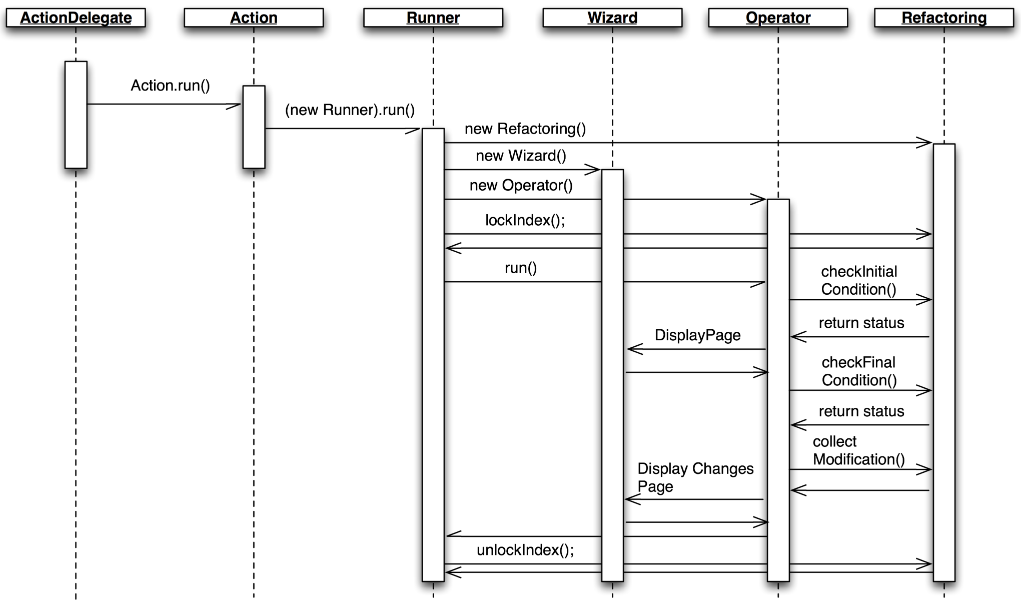


Figure 2.1.: Call sequence of a refactoring

2.1.1. ActionDelegate/Actions

Hook methods: run()
The *ActionDelegate* is linked to an element in the Eclipse GUI, for example a button or a menu item. On call, it invokes the *Action*, which will initiate the *Runner* (2.1.2) to execute the refactoring process.

2.1.2. Runner

Hook methods: run()
The *Runner* prepares the context for the actual refactoring components. When called, the *Runner* locks the index and invokes the main components of the refactoring which are:

- Operator
- Information class
- RefactoringWizard
- CRefactoring

After the refactoring is finished, the *Runner* will release the index.

2.1.3. Operator

Hook methods: Using framework implementation *RefactoringWizardOpenOperation*
The *Operator* opens the wizard dialog and calls `checkInitialConditions` (2.1.5.1) of *CRefactoring* (2.1.5). Depending on the success of this check, it will show an error message or the first page of the refactoring wizard.

2.1.4. Information class

Hook methods: Does not extend framework class
This class acts as a data transfer object (DTO). It stores all the information that has to be transferred from the wizard (2.1.6) to *CRefactoring* (2.1.5). This is where the refactoring options the user chooses are stored.

2.1.5. CRefactoring

Hook methods: checkInitialConditions(), checkFinalConditions(), collectModifications()
This class is the working horse of the whole refactoring process. It gathers information from the user's source code and will finally change it in the way the user specified in the wizard (2.1.6).
It mainly consists of three hook methods that will be called by the refactoring framework.

2.1.5.1. checkInitialConditions

This method is called by the operator (2.1.3) before the wizard (2.1.6) is displayed. As the name suggests, it checks if the actual state, meaning the state of the source code to be refactored, and the selection of the user are sufficient to perform the refactoring.

2.1.5.2. checkFinalConditions

This check is performed after the user has finished the last *UserInputWizardPage* (2.1.6.1). It verifies if the options selected by the user are applicable to the source code.

2.1.5.3. collectModifications()

In this function the AST (2.1.8) changes are made. Depending on the options the user specified in the *wizard* (2.1.6), it computes all node modifications on the AST (2.1.8) and stores them into a collector object received by call parameter. When finished, the framework will take over and present a preview of all code changes to the user. If he approves, the changes are applied to the code.

2.1.6. RefactoringWizard

Hook methods: addUserInputPages()
The refactoring framework provides a graphical user interface (GUI). Most parts of this wizard, such as the the error page (2.1.6.2) and the refactoring preview (2.1.6.3), are automatically generated by the LTK. A plug-in developer can add one or more *UserInputWizardPages* (2.1.6.1) to interact with the user. The used GUI framework is the Standard Widget Toolkit (SWT) [SWT].

2.1.6.1. UserInputWizardPage

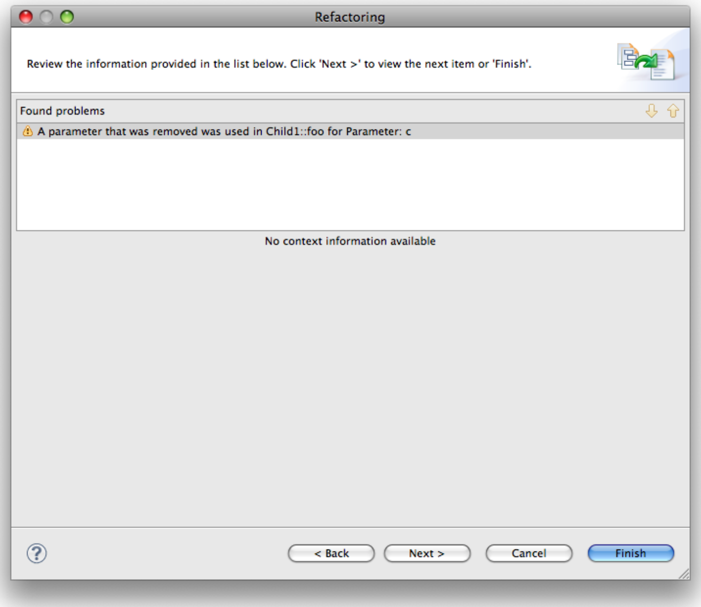
Hook methods: createControl()
These are the pages a plug-in developer can design to dynamically interact with the user, present information to him and offer options on how the refactoring should be applied.

2.1.6.2. Error page

If there are any errors within the *RefactoringStatus* object (2.1.7) after `checkFinalConditions` (2.1.5.2) has finished, the error page displays a list containing a short description of the errors in the top half of the frame and a more precise explanation of the selected error in the lower half (Figure 2.2).

2.1.6.3. Refactoring preview

When `collectModifications` (2.1.5.3) has finished, the refactoring framework processes all changes and computes the new state of the code. Before this is applied to the actual files representing the code, the refactoring preview displays all changes for the user to verify.



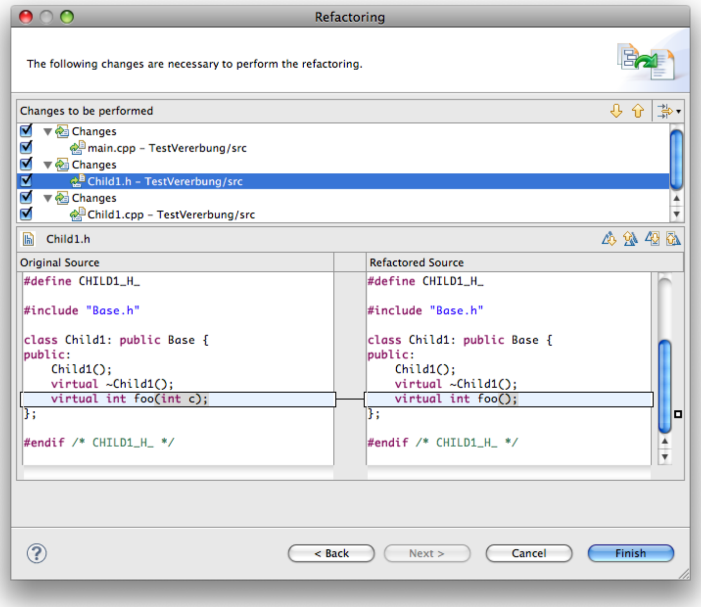
Review the information provided in the list below. Click 'Next >' to view the next item or 'Finish'.

Found problems

A parameter that was removed was used in Child1::foo for Parameter: c

No context information available

Figure 2.2.: LTK provided error page



The following changes are necessary to perform the refactoring.

Original Source:

```
#define CHILD1_H_

#include "Base.h"

class Child1: public Base {
public:
    Child1();
    virtual ~Child1();
    virtual int foo(int c);
};

#endif /* CHILD1_H_ */
```

Refactored Source:

```
#define CHILD1_H_

#include "Base.h"

class Child1: public Base {
public:
    Child1();
    virtual ~Child1();
    virtual int foo();
};

#endif /* CHILD1_H_ */
```

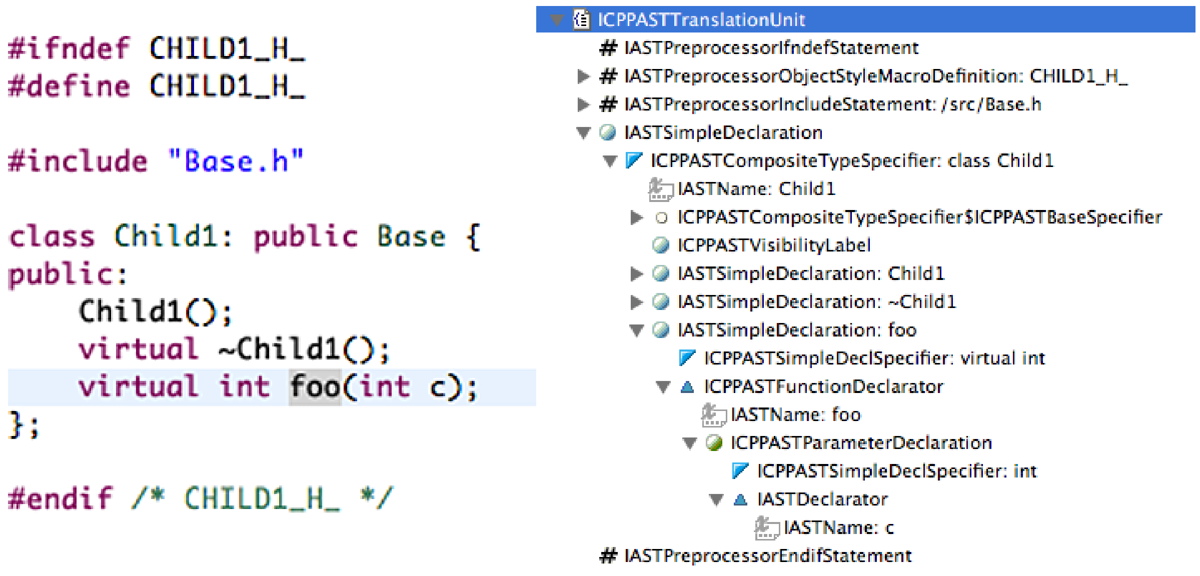
Figure 2.3.: LTK provided before - after page

2.1.7. RefactoringStatus

The status object contains all warnings and errors `checkInitialConditions` (2.1.5.1) and `checkFinalConditions` (2.1.5.2) detected. It is created in `checkInitialConditions` and is then given back as return type and handed over to `checkFinalConditions` as a parameter. Eventually, it will be passed on to the error page (2.1.6.2) of the wizard. It is not necessary for the returned object to be the one that's handed over, but it is strongly recommended, so that no errors or warnings get lost in the process.

2.1.8. Abstract Syntax Tree

The Abstract Syntax Tree (AST) is the representation of the source code provided to the refactoring by the LTK and the parser. During the refactoring process there is no need to manipulate any actual source code, all changes are made on the AST. Finally, the refactoring framework transforms the AST back to source code in the files using the *ASTRewriter* (2.1.8.2).
It is important to know, that any AST node created from the source code is read-only. In order to manipulate nodes, they have to be copied.



```
#ifndef CHILD1_H_
#define CHILD1_H_

#include "Base.h"

class Child1: public Base {
public:
    Child1();
    virtual ~Child1();
    virtual int foo(int c);
};

#endif /* CHILD1_H_ */
```

```
ICPPASTTranslationUnit
  IASTPreprocessorIfndefStatement
  IASTPreprocessorObjectStyleMacroDefinition: CHILD1_H_
  IASTPreprocessorIncludeStatement: /src/Base.h
  IASTSimpleDeclaration
    ICPPASTCompositeTypeSpecifier: class Child1
      IASTName: Child1
      ICPPASTCompositeTypeSpecifier$ICPPASTBaseSpecifier
      ICPPASTVisibilityLabel
      IASTSimpleDeclaration: Child1
      IASTSimpleDeclaration: ~Child1
      IASTSimpleDeclaration: foo
        ICPPASTSimpleDeclSpecifier: virtual int
        ICPPASTFunctionDeclarator
          IASTName: foo
          ICPPASTParameterDeclaration
            ICPPASTSimpleDeclSpecifier: int
            IASTDeclarator
              IASTName: c
  IASTPreprocessorEndifStatement
```

Figure 2.4.: Example of an AST - left side: source code, right side: parsed AST

2.1.8.1. Translation Unit

The translation unit (*IASTTranslationUnit*) is the root node in the Abstract Syntax Tree (AST) and therefore represents a parsed source file. When the parser is called repeatedly, it is possible to get multiple translation units for one source file. This can lead to runtime errors or undesired output, as changes of a file, based on different translation units, can produce conflicts due when file changes are made. It is therefore advisable to only use one translation unit per file. In the "Change Function Signature" refactoring (4) we avoided this situation by using a `HashMap`, with the file as key and the translation unit as value. That way, we were able to check whether a translation unit for the file already existed and directly use it if it did. Otherwise, we would parse the file once and then safe the translation unit into the hashmap.

Listing 2.1: Using the HashMap to get the translation unit
```
if (getTranslationUnit(file) != null) {
    unit = getTranslationUnit(file);
} else {
    unit = TranslationUnitHelper.loadTranslationUnit(file, true);
}
```

2.1.8.2. ASTRewriter

To manipulate ASTs (2.1.8) we use *ASTRewriters* provided by the framework. Its interface offers methods to insert new nodes into the AST as well as remove or replace them.

Inserting new nodes To insert a node into any kind of tree two things must be specified: The new node's designated parent and its position in relation to its future siblings. For this reason, the method `insertBefore` of *ASTRewriter* takes the nodes of the parent and of the sibling that should follow the new node, called the insert point, as parameters, while the insert point might be `null`, indicating the new node should be the last sibling.
If we wanted to insert a new function declaration in the following example (Figure 2.5) right after the visibility label, the parent node would be the *ICPPASTCompositeTypeSpecifier* and the *IASTSimpleDeclaration* the insert point.

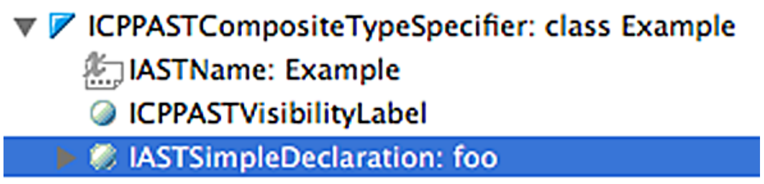

ICPPASTCompositeTypeSpecifier: class Example
IASTName: Example
ICPPASTVisibilityLabel
IASTSimpleDeclaration: foo

Figure 2.5.: Example of an AST - New node to be inserted above the selected one

2.2. Testing

As we intend to follow a test driven development [Beck02] approach, we will highlight the tools the refactoring offers and the strategy we developed to achieve a good test coverage.

2.2.1. Refactoring test framework

Since the refactoring highly depends on its context - there has to be a project, an index has to be built, a user selection must be available - it is not possible to check the interfaces with simple JUnit tests. We are therefore using a powerful test framework developed at the IFS [GrafButtiker06] designed to check the results of Eclipse refactoring plug-ins.

2.2.2. Test plug-in

The test process is located in its own plug-in to separate the test code from the actual refactoring code. We will now take a closer look at its components.

2.2.2.1. RefactoringTest

Hook methods: configureRefactoring(), runTest()
The test class plays a similar role in the test process as runner (2.1.2) in the refactoring process (Figure 2.1). Like the runner, it initializes *CRefactoring* (2.1.5), the information class (2.1.4) and starts the refactoring process.
Additionally, it can read properties out of a test file (2.2.2.2) and use them to configure the refactoring, simulating input the user would provide in the wizard (2.1.6), which can obviously not be used in an automated test.
When the refactoring process is done, the results are compared to the expected output specified in the test file.
Note that this test class provides a backbone for all refactoring tests. The actual tests are defined in the test files.

2.2.2.2. Test files

A refactoring always takes place within a context consisting of an Eclipse project, several files and a user input to start the refactoring. This context is created by the refactoring framework out of a test file following a specific structure.

Listing 2.2: Test file - Example.rts

```
//!Add a new int parameter
//define a default value for the callers
//#org.eclipse.cdt...ChangeFunctionRefactoringTest

//@.config
filename=Example.cpp
addParameter=int
addParamDefault=42
addParamDefaultInCaller=true

//@Example.h
class Example {
public:
    int /*$*/foo/*$$*/();
};

//=
class Example {
public:
    int foo(int new_int);
};
```

Explanation:

//!	Name of the test
//#	Fully qualified name of the `RefactoringTest` (2.2.2.1) implementation
//@.config	Start of the configuration section, followed by several property values
//@Filename	Listing of a file

//=	Expected state of the file after the refactoring
/*$*/Selection/*$$*/	Mark as user selected

Depending on the test, any number of files and their expected end state can be defined. Multiple tests can be defined in series in one test file.

2.2.2.3. Refactoring test suite

Hook methods: suite()
Extending JUnits *TestSuite*, this class collects different refactoring tests and can be called as a JUnit test. It is also possible to append normal JUnit tests.

Listing 2.3: Refactoring test suite - Adding two test files and a JUnit test

```java
public class ChangeFunctionTestSuite extends TestSuite {
   public static Test suite() throws Exception {
      TestSuite suite = new ChangeFunctionTestSuite();
      suite.addTest(RefactoringTester.
         suite("ChangeFunctionRefactoringTest",
            "resources/tests/AddParameter.rts"));
      suite.addTest(RefactoringTester.
         suite("ChangeFunctionRefactoringTest",
            "resources/tests/RemoveParameter.rts"));

      suite.addTestSuite(GUIStateTest.class);
      return suite;
   }
}
```

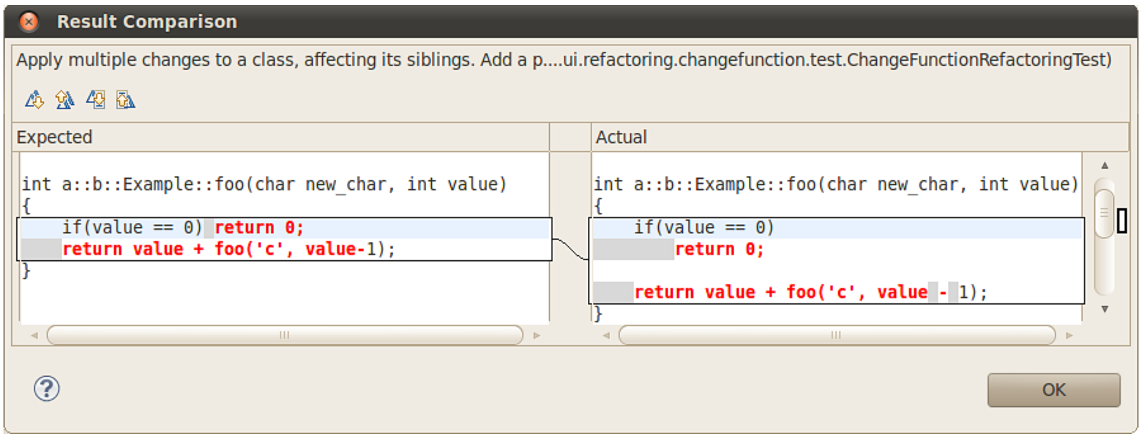
2.2.3. Test driven development

The concept of test driven development urges the programmer to first define the expected behavior of a class or method and write tests checking it before he actually starts implementing. This rule prevents the programmer from adept the requirements to the implementation instead of vice vercse. Additionally, he can easily check whether a change on existing code would destroy correct behavior simply by running the tests.

2.2.3.1. Problems

As mentioned before, a refactoring always takes place in a rather complicated context, which makes it almost impossible to just test an isolated interface. Most of the time, the only useful way is to check the output of the entire refactoring.
Defining the output is easy as we can specify what we want the code to look like in a test file (2.2.2.2). The problem starts, if we have to configure the refactoring to act as we want to. Since we are simulating the users input with properties in the configuration section of the test file, *RefactoringTest* (2.2.2.1) has to tell the refactoring how to behave, and therefore needs to have some information about how it is implemented.

An other problem is the way the actual output of the refactoring is compared to the expected one, defined in the test file. As the comparator checks for equality of each character, semantically equivalent pieces of code may fail the test.



Apply multiple changes to a class, affecting its siblings. Add a p....ui.refactoring.changefunction.test.ChangeFunctionRefactoringTest)

Expected:

```
int a::b::Example::foo(char new_char, int value)
{
    if(value == 0) return 0;
    return value + foo('c', value-1);
}
```

Actual:

```
int a::b::Example::foo(char new_char, int value)
{
    if(value == 0)
        return 0;

    return value + foo('c', value - 1);
}
```

Figure 2.6.: Semantically equivalent code fails test

2.2.3.2. Approach

As we can not consequently follow test driven development, we decided to make the following approach:

1. Choose the use case to be implemented
2. Write initial state in Eclipse CDT, make sure code is correct
3. Transfer initial state into *test file* (2.2.2.2)
4. Perform refactoring by hand, make sure code is correct
5. Transfer expected state into test file
6. Implement use case
7. Adept the implementation of *RefactoringTest* 2.2.2.1 to configure new functionality, if necessary
8. Correct whitespaces in test file to match refactoring output

2.3. PImpl tools

While we were planning the implementation of those new refactorings, we noticed that we have already solved many detail problems during our semester thesis. Here is an overview of what we could reuse.

2.3.1. NodeFactory

This factory offers some methods to create different AST (2.1.8) nodes out of simple parameters or other nodes. Especially `createDeclarationFromDefinition` and `createVisibilityLabel` where useful.

2.3.2. NodeContainer

The NodeContainer eases changing of the AST (2.1.8) as it keeps a node together with its ASTRewriter (2.1.8.2).

3. Declare Function

Unlike a Java method, a C++ function has to be declared before another function can call it. This can be done by simply defining the function before the caller in the source code, but more common is a separate declaration. Usually, this declaration is written into a header file so callers do not need to know the implementation of the function. Until now, the programmer had to write both the declaration and the definition, by hand. Our refactoring allows the programmer to write a function definition into the implementation file and then generates it's declaration based on this definition. It is the counter part to the already existing refactoring "Implement Methode" (sic) which creates a definition stub in an implementation file based on a method's declaration.

3.1. Essential use case: Declare a function

Goal: Declare a function in header file already used in this implementation file
Actor: User
Preconditions: The user selects a function definition without a corresponding declaration in a header file
Basic course of events:

1. The user selects a function and starts the "Declare Function" refactoring
2. A wizard (3.2.4) is presented to the user where he can select some options depending on the selected function
   Extensions:
   2.a) Declaring a free function (3.1.1)
   2.b) Declaring a member function (3.1.2)
3. A preview (2.1.6.3) of the refactored code is presented
4. The user confirms the changes
5. The changes are applied to the actual code

Postconditions: The function is declared in the header file the user has chosen

3.1.1. Declaring a free function

Goal: Declare a free function in a header file
Preconditions:

- The user selects a function definition without a corresponding declaration in a header file

- At least one function in the implementation file is declared in a header file. If there is more than one, a selection will be offered in the wizard (3.2.4).

Basic course of events: 2.a) The wizard presents a list of all header files, where another function in the implementation file is declared in. The user selects the desired target.
Extensions: 2.a)i) with a namepace (3.1.1.1)
Postconditions: The function is declared in the desired header file

3.1.1.1. With a namespace

Goal: Declare a free function defined in a namespace in a header file
Preconditions:

- The user selected an undeclared free function defined in a namespace
- At least on function within the implementation file is declared in a header file where the target namespace already exists

Basic course of events: 2.a)i) The wizard presents a list of all header files another function in the implementation file is declared in and that have the target namespace. The user selects the desired header file
Postconditions: The function is declared in the desired header file within the correct namespace

3.1.2. Declaring a member function

Goal: Declare a member function in the declaration of its class
Preconditions:

- The user selected an undeclared member function
- The target class is declared in a header file
- At least one member function of the target class is defined in the same implementation file and already declared

Basic course of events: 2.b) The wizard (3.2.4) presents a selection whether the visibility of the function should be public, protected or private
Postconditions: The function is declared in the class declaration with the desired visibility

3.2. Description of the refactoring code

In this section, we will explain how we implemented the elements of the refactoring process for our refactoring. For further information about the role of those elements in relation to the entire refactoring process, please consult the Language Toolkit (2.1) section.

3.2.1. checkInitialConditions

This method has several functions to prepare the refactoring and collect all the necessary informations.

1. Call `checkInitialConditions` of the super class.
2. Check if the selection is not in a header file.
3. Find the AST root node (2.1.8) of the function in which the selection is located.
4. Check if the selection has been parsed to `IASTSimpleDeclaration` due to not compilable code. If that is the case, create an equivalent definition node using the `NodeFactory`.
5. Search for all available header files.
6. Check if the chosen function is not already declared.
7. Determine if the function is in a class, a namespace or neither of the two.
8. Find possible points to insert the function declaration in the header.

To check if a translation unit (2.1.8.1) belongs to a header file, we simply read its `isHeaderUnit` flag.
To find the root node representing the selected function, we use a *CPPASTVisitor* to traverse the AST (2.1.8) until it finds a node of type *CPPASTFunctionDefinition* or *IASTSimpleDeclaration*.
If the root node is an *IASTSimpleDeclaration*, it has to be transformed into a *CPPASTFunctionDefinition* to be further processed. This task is performed by `createDefinitionFromDeclaration` (2.3.1).
The search for all header units is handled by `findHeadUnits` (3.2.1.1).
The test wether a function is already declared is done pretty simple: We resolve the binding of the function's *IASTName*. If we find something else, the selected function definition has been declared, in which case the refactoring will abort and show an error message.
If the function we want to declare is in a composite type or a namespace, we call the `getInsertParent` (3.2.1.2) method. It determines the right point to insert the new declaration.
We now have all information we can get from the selection and therefore know what to ask the user in the wizard.

3.2.1.1. findHeadUnits

The search for all available translation units (2.1.8.1) is done by iterating over all functions in the implementation file. For every function, we resolve the binding to its declaration in the header file and use that path to get the right translation unit (2.1.8.1). We save these units into the information class (2.1.4) so we can later choose in which one we want to insert the new declaration.

3.2.1.2. getInsertParent

In this method, we iterate over all nodes, that are possible candidates for being the parent node of the insert point (2.1.8.2), beginning with the translation units we found earlier. First, we split up the fully qualified name (Listing 3.1) of the function we want to declare and compare the first component to all namespace and class declarations of all translation units (2.1.8.1). Every match is stored in a candidate list. Then we search the next component in this list and repeat the iteration for every name component. In the end, we have a list of possible parents.

Listing 3.1: Example fully qualified name

```
void Namespace::SubNamespace::ClassName::FunctionName();
```

3.2.2. checkFinalConditions

`CheckFinalConditions` does nothing except for calling `super.checkFinalConditions`.

3.2.3. collectModifications

This method is called when the user has chosen the parameters in the wizard (3.2.4) and proceeds. It calculates all necessary changes to the header file using an *ASTRewriter* (2.1.8.2).
The operations inside `collectModifications` are structured as follows:

1. Create the AST node (2.1.8) representing the new declaration
2. Create a *NodeContainer* (2.3.2) for inserting that node
3. If the node is to be inserted into a composite type, find the appropriate visibility label or create one
4. Add the declaration node at the user selected insert point

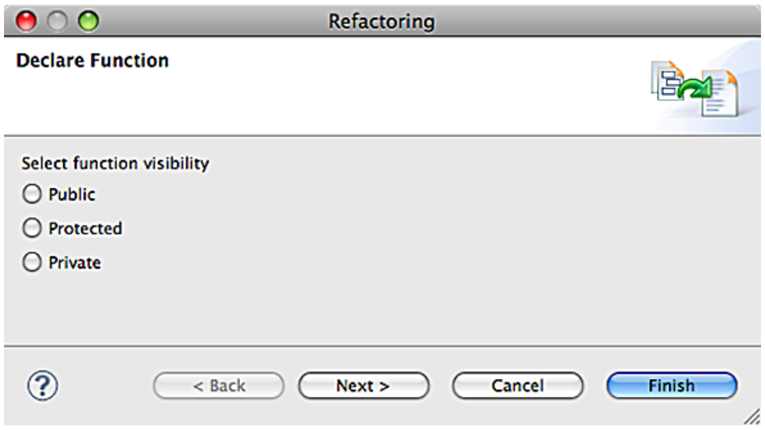
The new declaration of free functions is added at the end of the user selected header file or at the end of its namespace declaration, if applicable. Composite types are more difficult, since the right visibility label has to be used. This task was extracted into the `getInsertPointForClass` (3.2.3.1) method.
To create the declaration node, the *NodeFactory* (2.3.1) from the Introduce Pimpl Refactoring [PImpl09] is used.

3.2.3.1. getInsertPointForClass

This method finds the insert point (2.1.8.2) for the declaration inside a composite type declaration. First, it traverses all child nodes comparing all *ICPPASTVisibilityLabels* to the visibility the user has chosen. If a matching label is found, the next node is stored as insert point. If no matching label is found, a new one will be created using the *NodeFactory* (2.3.1) and will be inserted at the end of the declaration. The insert point for the function declaration will therefore be `null`.

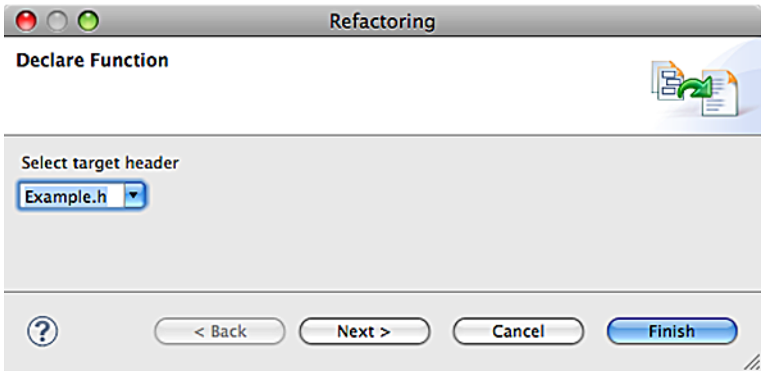
3.2.4. Refactoring wizard

If the selected function is not a free function and only one possible insertion point has been found, we have to interact with the user. So in most cases, a *UserInputWizardPage* (2.1.6.1) is added to the wizard (2.1.6).
If the function the user selected is a member of a composite type, the wizard lets the user choose the visibility of the declaration. Otherwise, he can choose in which of the found header files (3.2.1.1) the function should be declared.



Select function visibility
- ○ Public
- ○ Protected
- ○ Private

Figure 3.1.: Wizard: Choose the visibility for the declaration



Select target header: Example.h

Figure 3.2.: Wizard: Choose into which header file the function has to be declared

3.3. Testing

In this section, we will describe how we implemented the test plug-in (2.2.2) for "Declare Function".

3.3.1. DeclareFunctionRefactoringTest

The implementation of *RefactoringTest* (2.2.2) performs the "Declare Function" refactoring on test files (2.2.2.2). It needs to simulate the input a user would give by reading out properties (Table 3.1) from those test files. Since this refactoring offers very few options, only two properties are sufficient. We simply have to write those values to the information class (2.1.4)

Table 3.1.: Configuration properties for "Declare Function" test files

Property name	Allowed values	Function
visibility	"public"|"protected"|"private"	target visibility of a member function
unitNr	integer value ≥ 0	index value of target header file

3.3.2. Performed tests

Table 3.2.: Refactoring tests for "Declare Function"

Filename	Functionality to be tested
BasicTestFree.rts	Declare a free function in the only available header file
BasicTestClass.rts	Declare a member function in its class declaration
	public, protected and private
AdvancedTestFree.rts	Declare a free function into a selected header file
AdvancedTestClass.rts	Declare a member function in its class declaration
	Check correct detection of namespace insertion points

4. Change Function Signature

Changing the signature of an existing function is a very common refactoring. A developer may add an additional parameter to make his code adaptable, as he realizes his function could handle more problems that he initially designed it for or remove a parameter that turned out not to be used. He could want to generalize the parameters or, on the contrary, further specialize the return value. The imaginable possibilities of what a programmer might change are endless.
The sheer variation of possible changes make refactoring very challenging, especially when taken into account, that every change may cause further changes all over the code. The C++ concept of multiple inheritance additionally increases the level of difficulty.
In this chapter, we will explain what requirements we have defined for our refactoring, which concepts and problems we had to handle, and how the final implementation works.

4.1. Functional Requirements

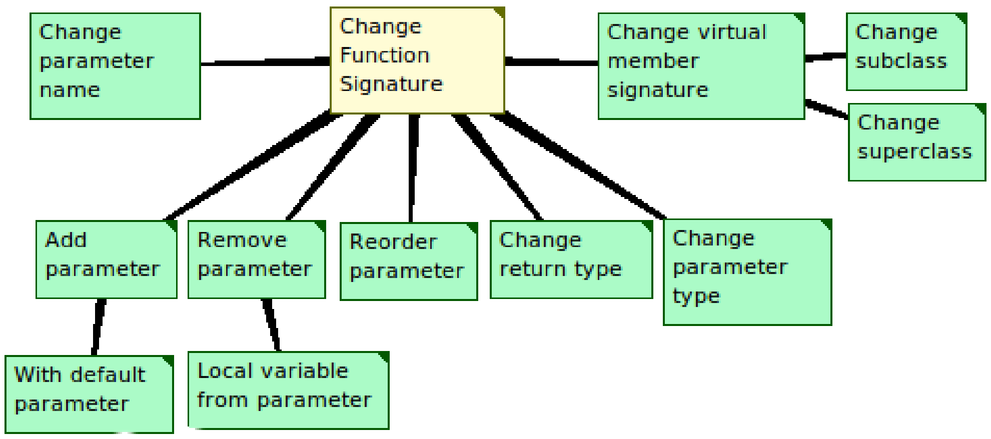


Figure 4.1.: Structure of the requirements

4.2. Essential use case: Change Function Signature

Goal: Change the signature of a selected function
Actor: User
Preconditions: The selected function must not have a predefined signature such as destructors or copy constructors
Basic course of events:

1. The user selects a function and starts the "Change Function Signature" refactoring

2. A wizard is presented to the user where he specifies the desired changes
   Extensions:
   2.a) Add parameter (4.2.1)
   2.b) Remove parameter (4.2.2)
   2.c) Reorder parameter (4.2.3)
   2.d) Change return type (4.2.4)
   2.e) Change parameter type (4.2.5)
   2.f) Change virtual member signature (4.2.6)
   2.g) Change parameter name (4.2.7)
3. Warnings are displayed if the refactoring could lead to incorrect code
4. A preview of the refactored code is presented
5. The user confirms the changes
6. The changes are applied to the actual code

Warnings: Several warnings might be presented, see sub use cases
Postconditions: The definition, all present declarations and all function calls, if possible and necesssary, are changed to match the new signature

4.2.1. Add parameter

Goal: Add a new parameter to the function
Basic course of events: 2.a) The user adds a new parameter and specifies its name and type
Extensions: 2.a)i) With default parameter (4.2.1.1)
Warnings: Calls will be extended with a placeholder which will lead to incorrect code
Postconditions:

- The definition and all declarations have been extended with the new parameter
- All function calls have been extended with a placeholder named like the parameter

Example:

Listing 4.1: Example: Add a parameter - before

```
int foo();
```

Listing 4.2: Example: Add a parameter - after

```
int foo(int i);
```

4.2.1.1. With default parameter

Goal: Add a new parameter with a default value to the function
Basic course of events: 2.a)i) The user specifies a default value for the new parameter. A checkbox offers the option to set the default value in the signature
Postconditions:

- The definition and all declarations have been extended with the new parameter
- A default value for the new parameter is defined within its declaration to its callers as parameter

Example:

Listing 4.3: Example: Add a parameter with default value in declaration - before

```
int foo();
```

Listing 4.4: Example: Add a parameter with default value in declaration - after

```
int foo(int i = 4);
```

Listing 4.5: Example: Add a parameter with default value in caller - before

```
int foo();

int bar(){
   return foo();
}
```

Listing 4.6: Example: Add a parameter with default value in caller - after

```
int foo(int i);

int bar(){
   return foo(4);
}
```

4.2.2. Remove parameter

Goal: Remove a parameter from a function
Preconditions:

- Function has at least one parameter

Basic course of events: 2.b) The user removes an existing parameter
Extensions: 2.b)i) Local variable from parameter (4.2.2.1)

Warnings: Show a warning if the removed parameter is used in the function and the refactoring will lead to incorrect code due to an undefined variable
Postconditions:

- The parameter is removed from the the definition and all declarations
- The parameter is removed from all calls

Example:

Listing 4.7: Example: Remove a parameter - before

```
int foo(int i);

int bar(){
   return foo(4);
}
```

Listing 4.8: Example: Remove a parameter - after

```
int foo();

int bar(){
   return foo();
}
```

4.2.2.1. Local variable from parameter

Goal: Transform the removed parameter into a local variable to get correct code after the refactoring
Basic course of events: 2.b)i) The user chooses the option to transform the parameter to a local variable
Postconditions:

- The parameter is removed from the definition and all declarations
- A matching local variable is declared and initialized at the beginning of the function definition
- The parameter is removed from all calls

Example:

Listing 4.9: Example: Remove a parameter and create a locale variable from it - before

```
int foo(int i){
   return i + 1;
}
```

Listing 4.10: Example: Remove a parameter and create a locale variable from it - after

```
int foo(){
    int i = 0;
    return i + 1;
}
```

4.2.3. Reorder parameter

Goal: Change the order of the parameters
Preconditions: The function has more than one parameter.
Basic course of events: 2.c) The user changes the order of existing parameters
Postconditions:

- The order of the parameters is rearranged in the definition and all declarations
- No parameter without a default value stands after any parameters with default values
- All calls have their parameters rearranged

Example:

Listing 4.11: Example: Reorder parameters - before

```
int foo(int i, Example &other);
```

Listing 4.12: Example: Reorder parameters - after

```
int foo(Example &other, int i);
```

4.2.4. Change return type

Goal: Change the return type of the function
Preconditions: Function has a return type
Basic course of events: 2.d) The user changes the return type of the function
Warnings: Show a warning if the new type can not be casted implicitly to the old and the refactoring will generate incorrect code
Postconditions:

- The return type of the signature is changed in the definition and all declarations

Example:

Listing 4.13: Example: Change return type - before

```
int foo();
```

Listing 4.14: Example: Change return type - after

```
double foo();
```

4.2.5. Change parameter type

Goal: Change the type of an existing parameter
Preconditions:

- At least one parameter

Basic course of events: 2.e) The user changes the type of an existing parameter
Warnings: Show a warning if the old type can not be casted implicitly to the new type and the refactoring will therefore generate incorrect code at the function calls
Postconditions:

- The type of the parameter is changed in the definition and all declarations

Example:

Listing 4.15: Example: Change parameter type - before

```
int foo(int i);
```

Listing 4.16: Example: Change parameter type - after

```
int foo(double i);
```

4.2.6. Change virtual member signature

Goal: Change the signature of a virtual member function
Preconditions:

- The selected member function is virtual

Basic course of events:

2.f) The wizard shows additional options for virtual member functions
Extensions:
2.f)i) Change superclasses (4.2.6.1)
2.f)ii) Change subclasses (4.2.6.2)

Warnings: Show warnings if sub- and/or superclasses are not to be changed and the refactoring would therefore generate incorrect code
Postconditions: Changes are applied to sub- and/or superclasses too if selected

4.2.6.1. Change superclasses

Goal: Apply the changes of a virtual member function to its superclasses
Basic course of events: 2.f)i) The user selects the option of changing the superclasses
Postconditions: The changes of the function signature are also applied to the function overriding it in its subclasses

4.2.6.2. Change subclasses

Goal: Apply the changes of a virtual member function to its subclasses
Basic course of events: 2.f)ii) The user selects the option of changing the subclasses
Postconditions: The changes of the function signature are also applied to functions overriding it in its superclasses

4.2.7. Change parameter name

Goal: Change the name of a parameter
Preconditions:

- At least one parameter

Basic course of events: 2.g) The user changes the name of an existing parameter
Postconditions:

- The name of the parameter is changed in the definition and in all declarations where the parameter had the same name before
- All occurrences of the parameter in the function body are adapted

Example:

Listing 4.17: Example: Change parameter name - before

```
int foo(int i);
```

Listing 4.18: Example: Change parameter name - after

```
int foo(int number);
```

4.3. Concepts and Problems

Until we had a working refactoring, there was a lot of planning and we made a lot of mistakes, so some of our design decisions had to be changed in the progress.

4.3.1. Signature abstraction

The AST tree is not designed to make major changes easily. Therefore, rearranging, adding or removing parameters in the AST signature is not a simple task. This fact in mind, we decided early in the development phase to create our own representation of a signature. This allowed us to manipulate everything in an easy way and not to be restricted by the AST or the number of changes. Our first idea was to record all the changes the user made and to apply them in the end to the AST tree. This plan was abandoned early, because the optimization problem turned out to be unnecessary complicated: For example, a ”move up” and a ”move down” change would cancel each other out. To avoid to develop an algorithm solving this optimization problem, we decided to look for a different approach.

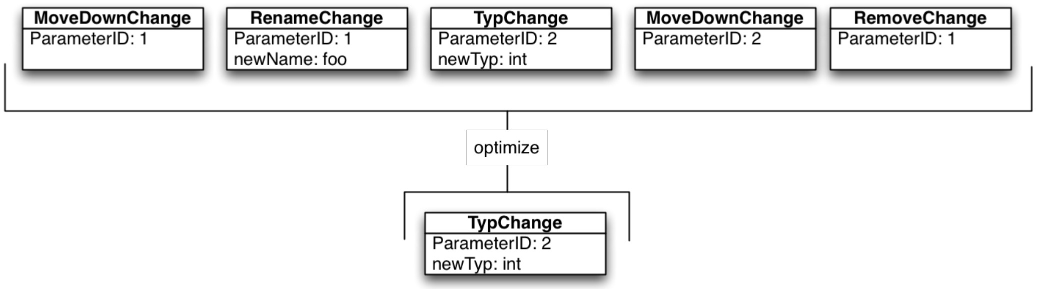


Figure 4.2.: The optimization problem here is, finding out that you do not need the *MoveDownChange* of parameter two

The plan was to create a representation of the signature in the form of strings that can be easily edited in the GUI and then parsed back to an *ASTNode* to be inserted back into the AST. Our goal was to create all the different definition, declaration, and caller nodes from the same signature.

4.3.1.1. Create Signature

To create a complete signature, we need information from a definition and from at least one caller. If we have neither, we just enter what we have. This is not a problem since we are not creating any new nodes. For example, if there are no declarations, we don’t need the information of the declarations because we don’t need to write any declaration back in the end.

4.3.2. Multiple Translation Units

During the development, we discovered some strange behavior when we had to handle forward declarations. If we made more than one change to a file, the offsets of the changes were messed up.

Listing 4.19: Add a parameter to a function that is forward declared as seen in Listing 1.2 the refactoring made a mess

```
int foo(int); // Forward declaration of foo
int bar(int, int number); // Forward declaration of bar

int foo(int i) {
   if (i == 0) return 1
   return fooint bar(int j, int number){
   if (i == 1) return 1
   return foo(i-1) / 2;
}(i) - 1;
}
```

At this point we realized that our source file was parsed multiple times. Therefore, every change was made in a separate translation unit. If one change adds or removes some characters, the position in the source code of the second change is not adjusted.
The solution to avoid this behavior was to make sure that every file is parsed just once into a translation unit (2.1.8.1).

4.3.3. Recursive Functions

A recursive function is a function that calls itself. This was a problem for us, because the call was found as caller but the body of a function is also a child of the function node. Therefore, the refactoring tried to change this part of source code twice witch lead to an exception.

Listing 4.20: Example of a recursive function: factorial

```
int fac (int nr) {
   if (nr = 0) return 1;
   return nr * fac(nr-1);
}
```

To solve this dilemma, we decided to skip the recursive calls in the process of handling callers and change them when we handle the function body. First, we checked if a caller was recursive during the handling of the change, but it turned out to be more intelligent to just check that when we add the caller to the caller list (4.4.9.8).

4.3.4. Handle Function Body

The function body is handled inside the creation of the new signature definition. We started out with just adding the old body to the definition node. But - as specified in the requirements (4.1) - if a parameter is renamed, it has to be renamed in the body too. Further, for every removed parameter a local variable should be initialized at the beginning of the function body. The renaming is of course only done for the function where the refactoring is called from, not for any inheritances that are also handled, since the parameter names do not matter concerning the differentiation of signatures.

4.3.5. Virtual member functions

In opposition to Java, dynamic polymorphism has to be explicitly marked for every function by using the keyword `virtual`. Important for us is the fact, that a virtual member function marks all equivalent functions in its subclasses, even if they are not marked as virtual in their own declaration. It is therefore not sufficient to simply check the presence of this keyword to find out if a function is virtual. We need to climb up its entire inheritance structure and check every occurrence of this specific signature to find out if one of its base classes introduces dynamic polymorphism.
As this virtual root function defines an interface for all its subclasses, it is in most cases utile to apply a change on all classes implementing it.
In consequence, we gather the entire inheritance tree of a selected function and present all found occurrences of its signature to the user, which can choose where the changes also should be applied to.

4.3.6. Using the parser

In our planning of the refactoring, we decided to use the parser late in the process and started out with some simple string to node parsing by hand. In retrospective, that was a very bad decision, since we could have saved a lot of work by just using the parser. With its help, we are able to generate all the necessary nodes right from strings, so we do not need to think about the creation of *IASTNodes* and can simply generate a string that looks right and get it handled by the parser. In the end, we decided anyways that not all nodes are created by the parser. The callers are simple structured nodes, so we decided to let them be created by hand, because with the parser we had to make a whole function around the callers and then extract the caller. This is more complex than generating the caller and replacing the original node.
The reason why we did not start earlier with the parser is because we were simply scared by such a big and complicate piece of software. For the next project, we will try using such existing help mechanics right from the beginning.

4.3.7. Errors and warnings

Some changes a user might want to apply can lead to compile errors. Those are:

- Use of an unknown type (4.3.7.1)
- Changing a type (4.3.7.2)
- Removing a parameter which is in use in the function body (4.3.7.3)
- Adding a new parameter without an appropriate default value (4.3.7.4)
- Multiple occurrences of the same parameter name (4.3.7.5)

There are two possibilities where to show these warnings: Our own input page (4.4.11.1) or the error page (2.1.6.2). The decision mainly depends on the point in the refactoring process where we can actually detect the possible problem.

4.3.7.1. TypeNotFound

Type: Warning
Detected by: *GUIState* (4.4.12)
Shown in: Input page (4.4.11.1)
This warning is created when the user enters a type that can not be verified.

4.3.7.2. TypeChanged

Type: Warning
Detected by: *GUIState* (4.4.12)
Shown in: Input page (4.4.11.1)
In the initial design, we planned to check if the new type can be implicitly casted to the old one for return or vice versa for parameter types. Unfortunately, we were not able to figure this out within real time, as it had to be done during user input, with acceptable reliability. So we decided to just warn the user.

4.3.7.3. RemovedParameterWasUsed

Type: Warning
Detected by: `checkFinalConditions` (4.4.5)
Shown in: error page (2.1.6.2)
If a removed parameter is used within the function body and the user does not choose the option to replace it with a local variable, it is of course undefined and the code can not compile.

4.3.7.4. PlaceholderWarning

Type: Warning
Detected by: *GUIState* (4.4.12)
Shown in: Input page (4.4.11.1)
If the user adds a new parameter and does not specify a default value, our refactoring will add the new parameter's name as a placeholder to the callers. If there is not an identically named variable defined at this code position by coincidence, this code will not compile.

4.3.7.5. DuplicateParameterName

Type: Error
Detected by: *GUIState* (4.4.12)
Shown in: Input page (4.4.11.1)
As the name of a local variable has to be unique identically named parameters will lead to compile errors. The wizard does not allow the user to continue as long as this error exists.

4.3.8. Conclusion

Creating a refactoring for C++ is not an easy task at all. The language is extremely potent and has a lot of options. But there are standards and we should have spent more time for a deeper analysis of the problems. This was revealed very clearly just before the end of the work when we implemented the warnings and error handling. We planned to warn the user if one class is not automatically casted into another one. For the simple types, that is not big a problem, but for classes and standard types like *std::string* or *std::vector*, that is way more complicated. To solve this, we had to use the compiler on our code. Since we were in the last week, we decided to drop this feature out of time concerns but also because starting the compiler and compile the code would slow down our refactoring even more.

4.4. Code description

For the refactorings, we had to write code at five points:

- checkInitialCondition
- checkFinalCondition
- collectModifications
- Wizard
- Info Class

To make it easier for us, we added several helper classes and an abstract representation of the source code. The first thing we did, was introducing *Signature* as a representation of a signature and the *SignatureFactory*, so we could reuse everything in a later refactoring. In the end, we refactored a lot of code and extracted all methods covering the AST and resolving bindings into a separate *ASTHelper* (4.4.9) class.

4.4.1. Signature

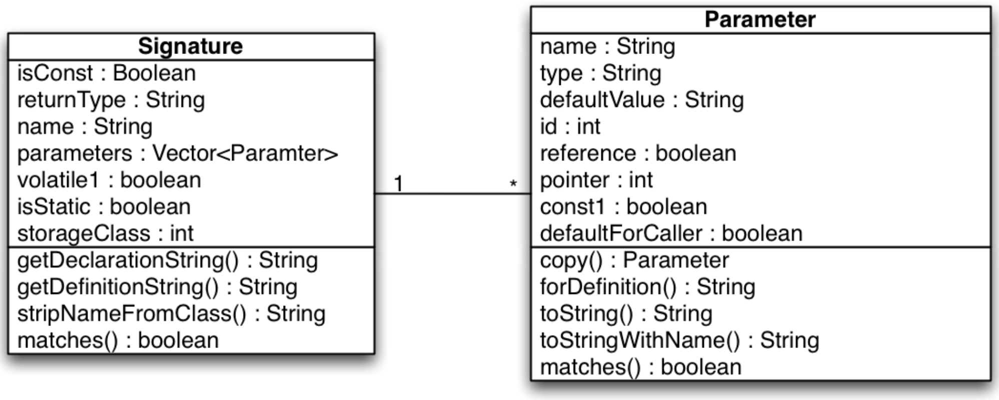


Figure 4.3.: Signature and Parameter

The signature is designed to represent all the information needed to create a definition, declarations and callers. Most things are saved as strings or other simple types. Only for the parameters, we made a separate class (4.4.2). At first, we planned to store the parameters in two vectors, one with the normal parameters and one with parameters with default values. This assured that the default values are always after the normal parameters. But this was unnecessarily complicated, therefore we switched to just one vector.

4.4.1.1. getDefinitionString

Returns a string that represents the signature of a definition. This can be parsed into a *ICPPASTFunctionDefinition* and then supplemented with a function body.

Listing 4.21: Example: Definition string

```
int Example::foo(Example &other, int i)
```

Parameter: ArrayList<String>parameterNames These are the original names the parameters have in their definition and declaration nodes. It is used for all signatures, except the pair where the refactoring is called from. If it is null, `getGeneralString()` is called and the names stored in the *Signature* are used.

Parameter: String name This is the fully qualified name used for the definition

4.4.1.2. getDeclarationString

Returns a string representation of a declaration. This can be parsed into an *IASTSimpleDeclaration*.

Listing 4.22: Example: Declaration string

```
virtual int foo( Example &other, int i = 4)
```

Parameter: ArrayList<String>parameterNames These are the original names the parameters have in their definition and declaration nodes. It is used for all signatures, except the pair where the refactoring is called from. If it is null, the names stored in the *Signature* are used.

Parameter: ArrayList<String>defaultValues A list with the default values extracted from the declaration it is supposed to replace. If this parameter is null, the default values from the *Signature* are used.

4.4.1.3. getGeneralString

This method returns a string representation of the definition as stored in the *Signature*.

4.4.1.4. stripNameFromClass

This is a static helper function that strips the class and namespace names from a string. It looks for the last occurrence of "::" and returns the substring after that.

Parameter: String name2 The fully qualified name, where we want to extract the function name from.

4.4.1.5. matches

Checks whether another *Signature* matches the interface of this *Signature* instance.

Parameter: Signature other The *Signature* which is checked against the instance.

4.4.1.6. compareParametersWith

A helper method that takes a vector of *Parameters* (4.4.2) and compares it with the instance's own *Parameters* using the `matches` (4.4.2.4) comparison method of *Parameter*.

4.4.2. Parameter

This class represents a parameter of a function. It is built the same way than the *Signature* with the possibility to return a string representation that is used by the *Signature* to return the strings that are parsed.

4.4.2.1. copy

Returns a deep copy of the *Parameter*

4.4.2.2. forDefinition

Returns the parameter with all the necessary information for a definition as a string.

Listing 4.23: Example: Parameter string

```
int i
```

Parameter: String parameterName The name of the parameter

4.4.2.3. toString

Returns the parameter with all the the necessary information for a declaration as a string.

Listing 4.24: Example: Parameter string

```
int i = 4
```

Parameter: String name The name that the parameter should have. If name is null, the name from the *Parameter* is used

Parameter: String defaultVal The default value that the parameter should have. If defaultVal is null, the value form the *Parameter* is used.

4.4.2.4. matches

This method compares another *Parameter* with itself to find out if their types are matching. Only `const1`, `reference`, `pointer` and `type` are compared.

Parameter: Parameter other The *Parameter* to be checked against.

4.4.3. Caller

The caller object represents a caller. We use it to simplify the modifications and the creation of the new caller nodes. The constructor takes an *ICPPASTFunctionCallExpression* and extracts all the information in it.

4.4.4. checkInitialConditions

This is one of the main functions of every refactoring. In our case we gather all the information we need to do the refactoring. Therefore, it is divided into several steps.
First, we decide if we start at a function definition or a declaration. From there on, we search all other declarations of this function and, if we started from a declaration, also the definition. Then we look for the callers in all related projects.
We also need to determine if we are in a class and find its inheritance structure.
From all we found, a *Signature* is created and saved into the info object.

4.4.4.1. findFirstSelectedNode

This method gets us the selected node in the AST tree. We use a visitor to traverse the whole AST and check if the selection is on the node we are checking. This simple function was already used in our older refactoring "Introduce PImpl" [PImpl09]

4.4.4.2. findFunctionNode

The `findFirstSelectedNode` method finds just any node inside a function that is selected. Here, we simply travel upward the tree until we hit a *ICPPASTFunctionDefinition* or a *IASTSimpleDeclaration*.

4.4.4.3. findClass

Based on the already found function declaration, this function checks if this declaration is part of a composite type declaration by checking whether its parent node (2.1.8) is a `ICPPASTCompositeTypeSpecifier` and stores this node to the *information class* (2.1.4) if true.

4.4.4.4. findInheritance

Because it is not sufficient to check for the presence of the keyword `virtual` (4.3.5), this method is looking for the topmost baseclass introducing virtual. If found, a *FunctionContainer* (4.4.8) is created and stored as virtual root node in the *information class* (2.1.4) and eventually a tree of the inheritance structure is builded out of all its subclasses.

4.4.5. checkFinalConditions

In this refactoring, the `checkFinalConditions` has not much to do because most of the problems are checked in the GUI as the user is making his changes. The only thing we look for here is, if a removed parameter was used in the function body.

4.4.5.1. checkForUsedParameter

First we have to know what parameters actually are removed. The `getRemovedParameters` methode provides this function. As soon as we know what parameters are removed, we us a visitor to traverse the function body and search for these parameters. If one is found, we add a warning to the status.

Parameter: RefactoringStatus status The status object where the warning should be stored

Parameter: final ICPPASTFunctionDefinition definition The definition that should be checked for a removed parameter that is used

4.4.6. collectModifications

`collectModifications` is divided in three similar parts. First it handles all callers, then all definitions and in the end the declarations.
All these three parts have the same course of events:

1. Iterate over a list of nodes that have to be replaced.
2. Create the new node with help form the *SignatureFactory*.
3. Create a *ASTRewriter* for the translation unit.
4. Create a *NodeContainer* where the changes are made.
5. Replace the old node with the new one.

4.4.7. SignatureFactory

In the beginning, the signature factory was a static class for creating signatures out of *ASTNodes* and vice versa. Before we started using the parser to create *ASTNodes*, it was a class with very long and complex methods. With the introduction of the parser, these methods got a lot shorter and more efficient.

4.4.7.1. createSignatureFromAST

This method is divided into two parts. First, all the information we can get out of the definition is collected, afterwards we search for the still missing information pieces like default values or flags, such as virtual, in the declaration.
From a programmers perspective, this method is not very difficult. There are just two kinds of problems here. First and foremost, the whole inheritance tree of the *ASTNodes*. Most of the time, you get an *IASTNode* or something similar from a method and after an `instanceof` check, you have to cast it into a specific *ICPPASTNode*. This makes the

code look way more complex than it actually is. The methods we do not mention here are products of refactoring, which encapsulate such casts in separate methods. The other problem was to get all the information out of the declaration and definition, since some things are saved in one of them and other times in both. This requires to pay attention especially if you have only one.

Parameter: IASTNode selectedDefinition The definition the user wants to refactore.

Parameter: IASTNode selectedDeclaration The declaration the user wants to refactore.

4.4.7.2. definitionToSignature

Here we get all the information possible out of a definition:

- Function name
- Constant function
- Volatile function
- Return type
- Parameters via call of `extractParameters`
  - Parameter name
  - Parameter type
  - Constant parameter
  - Reference parameter
- Storage class

Parameter: IASTNode selectedDefinition This is the definition node from where the information has to be extracted.

Parameter: Signature result This is the signature object that everything has to be saved within.

4.4.7.3. declarationToSignature

All information available in a declaration are extracted here.

- Storage class
- Constant function
- Volatile function
- Parameters via call of `extractParameters`
  - Parameter name

  - Parameter type
  - Constant parameter
  - Reference parameter
  - Default value
- Function call to `getInfoFromDeclSpecifier`
  - Function return type
  - Virtual member function

Parameter: IASTNode selectedDeclaration This is the declaration node from where the information has to be extracted.

Parameter: IASTNode selectedDefinition It checks if there was already a definition read out. If this parameter is not null, it does not read out the information already gathered from the definition.

Parameter: Signature result This is the signature object that everything has to be saved within.

4.4.7.4. extractParameters

This function first checks if all parameters are already in the signature. If that is the case, it just adds any default value that is there, otherwise it calls `getNewParameters`.

Parameter: Signature result This is the signature object that everything has to be saved within.

Parameter: IASTParameterDeclaration[] parameters An array of *ASTNodes* of the parameters, which have to be converted into *Parameter* (4.4.2).

4.4.7.5. simpleTypeToString

The C++ parser differentiates two kinds of types. On one hand we have simple types and on the other named types. Simple types are the built-in types:

- bool
- char
- double
- float
- int
- void (although this is not an actual type but for the AST and therefore for us it's handled like a simple type)

Named types are everything else. For us, this is important, since these are two different *ASTNodes* we have to handle. In the named types, the type is stored as an expression where we can extract the string without any problem. For the simple types, it is more complicated, since they just save an integer to determine what kind of type they are. Of course, the user does not want to see this numbers, but the actual type names. We made this function to convert from those integers to type strings.

Parameter: ICPPASTSimpleDeclSpecifier cppSimpleDecl The simple type node, that has to be converted in a string.

4.4.7.6. createDeclarationASTNodeFromSignature

This is one of the three complementary methods to `createSignatureFromAST`. Its goal is to create an AST tree that represents a declaration in the source code from a *Signature* (**??**) and the declaration node it has to replace.
In the first version, we created all nodes by hand similar to the way we used in the "Introduce PImpl" refactoring [PImpl09]. After we got the parser running, the method was significantly reduced in its size and complexity. It now gets the declaration string out of the *Signature* and feeds it to the parser.

Parameter: SignatureChange sigChange The *Signature* object, from whom the new declaration is made, is stored in the sigChange parameter.

Parameter: IASTSimpleDeclaration oldDeclaraton The Node to be replaced with the new declaration. It is needed for the parameter names and default values of all declarations other than the one where the refactoring was called from.

Parameter: boolean isFirst Is true if the declaration is the one where the refactoring is called from.

4.4.7.7. createDefinitionASTNodeFromSignature

This method creates the signature of the definition similar to `createDeclarationASTNodeFromSignature`. The difference to the method above is the body it has to handle. This is extracted to `handleBody`.

Parameter: SignatureChange sigChange The *Signature* object, from whom the new definition is made, is stored in the sigChange parameter.

Parameter: IASTNode oldSignature To old definition, that is intended to be replaced with the new one. It is needed for the parameters of all definitions other than the one where the refactoring was called from.

Parameter: boolean isFirst Is true if the definition is the one where the refactoring is called from.

4.4.7.8. handleBody

In this method, two things are taken care of. With a visitor, the tree of the function body is traversed. If it wanders across an expression, it either does one of two things, depending if it is an *IdExpression* or a function call.
IdExpressions are handled just for the function, from where the refactoring is called, to adapt to the new parameter names. Recursive calls have to be handled for all definitions in the inheritance tree. They are changed the same way as all the other callers are handled

4.4.8. FunctionContainer

The idea of the *FunctionContainer* is to keep all AST (2.1.8) nodes related to a certain function definition in a single class. This includes the function definition, all declarations, and all callers.
If it is a member function, it also holds the node of the parent composite type and has a *FunctionContainer* list to store its direct subclasses. This offers the possibility to build a tree structure, especially to represent the inheritance structure (4.4.4.4).

4.4.9. ASTHelper

Into the ASTHelper class, we extracted some often used operations related to the AST (2.1.8). It also holds a `HashMap` connecting different source files to their translation unit (2.1.8.1). By consequently checking this map first before parsing again *translation units*, we avoid errors in the *ASTRewriters* (2.1.8.2) that occur, if a source file is manipulated based on different translation units.

4.4.9.1. findClassByName

This method searches for the *ICPPASTCompositeTypeSpecifier* of the declaration linked to the passed *IASTName*.

Parameter: IASTName className: The name representation of the composite type to be found.

4.4.9.2. typeExists

Implements a simple check whether a type with a specific name exists.

Parameter: IASTName typeName: The name representation of the type

4.4.9.3. getIFile

Returns the *IFile* representation of the source file associated with the passes AST node.

Parameter: IASTNode node: The AST node of which the *IFile* should be found

4.4.9.4. findSubClassesOf

As subclasses are not directly evident from the declaration, as baseclasses are, this method searches them.

Parameter: ICPPASTCompositeTypeSpecifier theClass: The baseclass of which the subclasses should be found

4.4.9.5. findDefinition

Finds the definition of an AST node.

Parameter: IASTNode selectedNode: The node whose definition should be found

Parameter: IIndex index: The index in which to search

4.4.9.6. findDeclarations

Finds all declarations of an AST node.

Parameter: IASTNode selectedNode: The node whose declarations should be found

Parameter: IIndex index: The index in which to search

4.4.9.7. findCalls

Finds all expressions calling the passed AST node.

Parameter: IASTNode selectedNode: The node whose callers should be found

4.4.9.8. isRecursive

This helper method checks if call expression within a function is a recursive call to itself.

Parameter: IASTCallExpression call: The caller expression

Parameter: ICPPASTFunctionDefinition definition: The function definition

4.4.10. ParseHelper

At different points in the refactoring process we need to parse strings as C++ code to check validity of user entered types or to create AST nodes. As the parser has to be configured to the projects context, this has been refactored to a class, which is attached to the information file (2.1.4) and therefore accessible during the entire refactoring process.

4.4.10.1. parse

Generates a `CodeReader` on the passed string, parses it and returns the generated AST node (2.1.8)

Parameter: String expression The string expression to be parsed

4.4.11. Wizard

As mentioned, "Change Function Signature" offers a variety of configuration options. While most refactorings just offer option flags and a fixed number of input, it has to handle a list of parameters and offer multiple options to manipulate them. Additionally, it has to continuously check the user's input for possible errors (4.3).
To both implement this functionality and keep the refactoring testable, we decided to extract all logic from the wizard into an additional class called *GUIState* (4.4.12) and let the wizard page just forward the user input to this class.

4.4.11.1. ChangeFunctionInputPage

As the logic has been extracted, *ChangeFunctionInputPage* does not much more than creating all GUI elements and creating handlers to catch user input, forward it to *GUIState* (4.4.12) and update the view.

4.4.12. GUIState

This class represents the state of the GUI as it is represented to the user. It also handles all user input and generates the new *Signature* (4.4.1)

4.4.12.1. ParameterRow

This container class is a row in the table representation of the *Parameters* on the input page (4.4.11.1). Besides the *Parameter*, it has an "active" flag and references to some GUI elements. These are important to change the visualization of the row.

4.4.12.2. setParameterDefaultValue

Sets the default value of the selected parameter. Calls `reorderParameters` (4.4.12.11).

Parameter: int row The index of the parameter to be changed

Parameter: String text The new default value

4.4.12.3. setParameterType

Sets the type of the selected parameter. Calls `checkParameter` (4.4.12.9).

Parameter: int row The index of the parameter to be changed

Parameter: String text The new parameter type

4.4.12.4. setParameterDefault

Changes the selection flag. Calls `reorderParameters` (4.4.12.11).

Parameter: int row The index of the parameter to be changed

Parameter: New value for the flag

4.4.12.5. addParameter

Adds a new *Parameter* (4.4.2) to the *GUIState*. Calls `reorderParameters` (4.4.12.11) and `checkParameter` (4.4.12.9).

Parameter: Parameter newParameter The new *Parameter*

4.4.12.6. setReturnType

Defines a new return type for the function. Calls `checkReturnType` (4.4.12.10).

Parameter: String text The new return type

4.4.12.7. moveUp

Moves the selected *Parameter* (4.4.2) up in the list. Calls `reorderParameters` (4.4.12.11).

Parameter: int row The index of the parameter to be moved up in the list

4.4.12.8. moveDown

Moves the selected *Parameter* down in the list. Calls `reorderParameters` (4.4.12.11).

Parameter: int row The index of the parameter to be moved down in the list

4.4.12.9. checkParameter

Checks if the parameter is valid in its context.

Parameter: ParameterRow parameterRow The *ParameterRow* to be checked

4.4.12.10. checkReturnType

Simply checks if the new entered exists using `typeExists` (4.4.9.2) and creates a warning if not and if it differs from the original return type of the function.

Parameter: String text The new return type

4.4.12.11. reorderParameters

Parameters having a default value defined in their declaration have to be at the end of the list. This method traverses all *ParameterRow* list and moves those to the back.

4.5. Testing

"Change Function Refactoring" was much more challenging to test than "Introduce PImpl" and "Declare Function", because of the complexity of possible user input.
To achieve a good test coverage, we separated the logic from the wizard and introduced *GUIState* (4.4.11). The idea is, that we can simulate the user input by calling the same methods of *GUIState* as the wizard does.

4.5.1. ChangeFunctionRefactoringTest

As there are many possible refactoring options, the list of configuration properties is rather long.

Table 4.1.: Configuration properties for "Change Function Signature" test files

Property name	**Allowed values**	**Function**
addParameter	a type name	adds a new parameter of the specified type
addParamDefault	a *String*	default value for the new parameter
addParamDefaultInCaller	true|false	add the default value to the callers
rmParamAtIndex	integer value ≥ 0	remove parameter on this index
changeReturnTypeTo	a type name	changes return type
moveParameterUpAtIndex	integer value ≥ 0	move up parameter at this index
moveParameterDownAtIndex	integer value ≥ 0	move down parameter at this index
changeParameterTypeAtIndex	integer value ≥ 0	change parameter type at this index
changeParameterType	a type name	new parameter type at index specified by changeParameterNameAtIndex
changeParameterNameAtIndex	integer value ≥ 0	change parameter name at this index
changeParameterName	a *String*	new parameter name at index specified by changeParameterNameAtIndex
changeDefaultValueAtIndex	integer value ≥ 0	change parameter's default value at this index
changeDefaultValue	a *String*	new default value at index specified by changeDefaultValueAtIndex
toggleDefaultInCallerAtIndex	integer value ≥ 0	switches default value position between callers an declaration
applyInheritance	true|false	if refactoring should be implied to entire inheritance structure

4.5.2. applyTestConfig

Our idea for testing is to manipulate *GUIState* (4.4.12). As this class is initialized during `checkInitialConditions` (4.4.4) we apply the refactoring options between `checkInitialConditions` and `checkFinalConditions` (4.4.5) just as the wizard would.

4.5.3. TestConfig

As the configuration properties (Figure 4.1) are read out before the actual refactoring process is started, we have to temporarily store them. For this task, we introduced the storage class *TestConfig*.

4.5.4. Performed tests

Table 4.2.: Refactoring tests for "Change Function Signature"

Filename	**Functionality to be tested**
AddParameter.rts	Adding a new parameter
ChangeParameterName.rts	Change the name of an existing parameter
ChangeParameterType.rts	Change the type of an existing parameter
ChangeReturnType.rts	Change the return type
Combination.rts	Many changes, with inheritance and namespaces
DefaultValues.rts	Correct handling of new and existing default values
DifferentCallers.rts	Detection and manipulation of callers at different places
DoNothing.rts	Does not apply any changes
HandleForwardDeclarations.rts	Correct adjustment of forward declarations
HandleInheritance.rts	Apply changes to member functions implementing the same virtual signature
Inline.rts	Correct manipulation of inline implemented member functions
Namespaces.rts	Handling different namespaces
RemoveParameter.rts	Remove a parameter
ReorderParameters.rts	Correct reordering of parameters in case of default values in the declaration

5. Result

During this bachelor thesis we succeeded in different goals. First of all we added two more refactorings to the palette of CDT, one of whom is very mighty. For us, "Change Method Signature" is a very important refactoring in Java, therefore we are glad to provide this tool to the C++ community as well. Apart from the products we created, we tried to improve a lot of things we had not done well during our semester thesis. This time, we had a running build server and automated testing for our code as planned. In the end, we understood, that having a build environment was worth all the hard work we had to invest in getting it up and running. We are now looking forward to hopefully getting our refactorings from these two theses integrated in the official eclipse build.

5.1. Declare Function

Declare function was planned as a warm up from the start. We had a lot of tasks on our hands with the build process so we were glad to have a something less complex as "Change Function Signature" to begin with. Even though we used the LTK for the "Introduce PImpl" refactoring, we had to work our way back into the topic. There have been some unexpected problems, but in the end we succeeded in finishing a small but useful refactoring, that redeems the software developer from some dull work tasks.

5.2. Change Function Signature

The main workload was by far the "Change Function Signature" refactoring. It was more complex than the rest of refactorings we had done so far. The most challenging part was to cover as many variations of C++ as possible. In the end, we could handle most code and produce the changes the user expects.
We were able to fulfill all the requirements specified at the beginning. After some design and coding work, we only had to drop some warnings (4.3.7)

5.3. Perspective

We are glad we could contribute our part, but there is still a lot of work left in order to make programming C++ in CDT as comfortable as programming Java is in JDT.

5.3.1. Improve "Change Function Signature"

The wizard could offer some comfort features, such as autocompletion or a selection pop-up for types.
The detection of possible errors after some refactoring options could be more precise, telling the user exactly whether the code will compile after the refactoring or not.

5.3.2. More refactorings

Many more refactorings could further ease programming C++. Such as:

- Move a function to another class
- extract an inline function
- inline
- extract superclass

5.4. Thanks

Our most important source of information was without any doubt Emanuel Graf. He was very helpful and willing to answer questions regarding the refactoring framework and CDT any time. To get the build process and the automated testing running, Lukas Felber was an irreplaceable resource. We also like to thank Marcel Huber and Nicolas Bigler for their effort in testing the refactoring from two different points of view. And then, of course, to our advisor Prof. Peter Sommerlad for his support and critics of our work.
We like to thank all these people for supporting our bachelor thesis.

A. User guide

Installing and using our refactorings is pretty easy. Just follow the instructions below.

A.0.1. Requirements

In order to use our refactorings, you need to have Eclipse 3.5.2 and CDT 6.0.2 installed. It might run on newer version, but it is not tested to do so.

A.0.2. Installation

To install the refactorings, unpack the zip files into the folder where your Eclipse CDT is located.

A.0.3. Use "Declare Function"

To use the "Declare Function" refactoring:

1. Select a not yet declared function.
2. Select "Declare Function" from the sample menu.
3. Select options:
   - If you have a class, choose the visibility.
   - If you have multiple header files, choose the target.
4. Finish the refactoring.

A.0.4. Use "Change Function Signature"

To use the "Change Function Signature" refactoring:

1. Select a definition or a declaration in one of the following views:
   - Editor
   - Outline
   - Project Explorer
   - C/C++ Projects
2. Select "Declare Function" from the sample menu.
3. Use the GUI:
   - Change return type of function
   - Set member function const

- Add parameter
- Remove parameter
  - Generate local variable from removed parameter
- Reorder parameter
- Manipulate parameter
  - Change parameter type
  - Set parameter const
  - Set parameter as reference
  - Change parameter name
  - Change default value
  - Set default value into caller

4. Finish the refactoring.

A.0.5. Use the plug-in in the final version

Our goal is to have our refactorings included in the next official release (2011) of CDT. They would then be accessible through the standard refactoring menus.

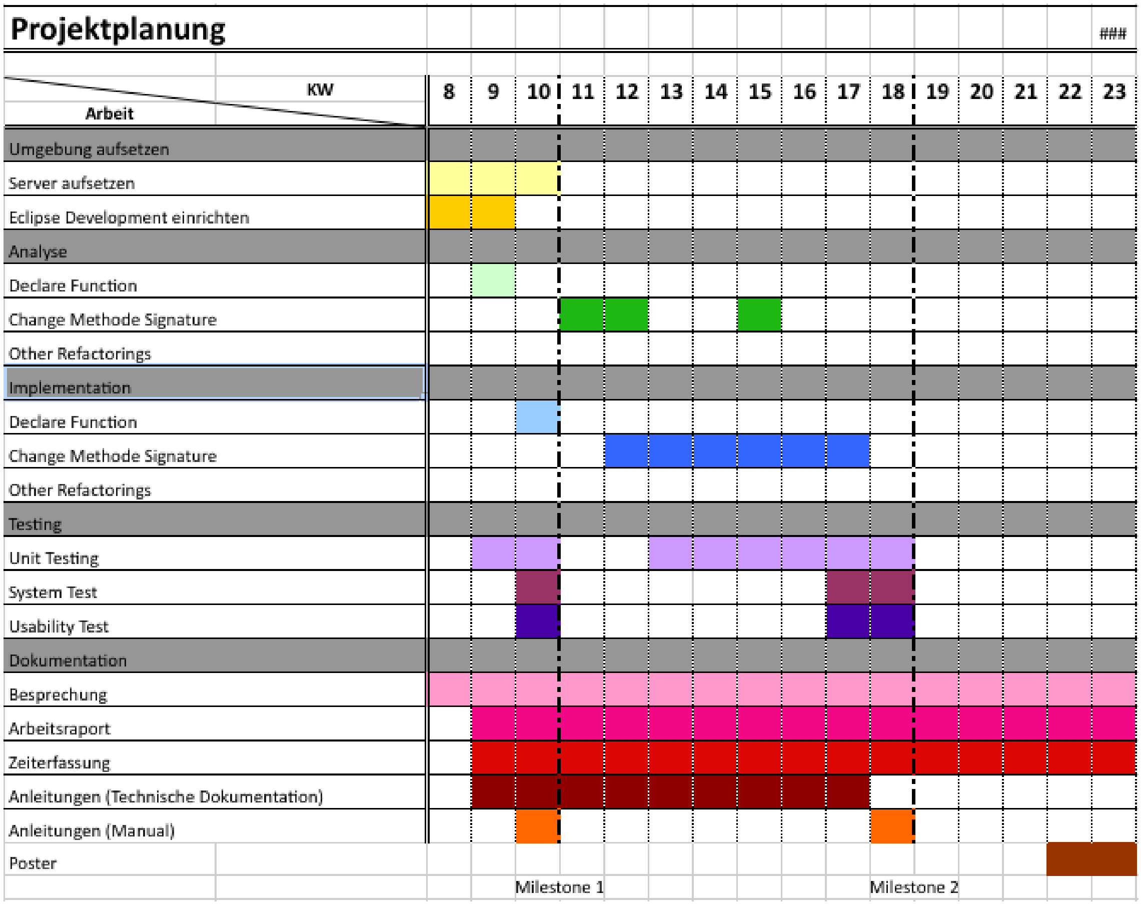
B. Management

B.1. Team

Matthias Indermühle
Signature, AST to Signature, Signature to AST

Roger Knöpfel
GUI, Buildserver, Parser, Testing

B.2. Project Plan



Projektplanung																###
KW / Arbeit	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Umgebung aufsetzen																
Server aufsetzen	🟨	🟨	🟨													
Eclipse Development einrichten	🟧	🟧														
Analyse																
Declare Function		🟩														
Change Methode Signature				🟩	🟩			🟩								
Other Refactorings																
Implementation																
Declare Function			🟦													
Change Methode Signature					🟦	🟦	🟦	🟦	🟦	🟦						
Other Refactorings																
Testing																
Unit Testing		🟪	🟪			🟪	🟪	🟪	🟪	🟪	🟪					
System Test			🟥							🟥	🟥					
Usability Test			🟪							🟪	🟪					
Dokumentation																
Besprechung	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Arbeitsraport		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Zeiterfassung		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Anleitungen (Technische Dokumentation)		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥						
Anleitungen (Manual)			🟧								🟧					
Poster															🟫	🟫
			Milestone 1								Milestone 2					

Figure B.1.: The initial project plan

We started as planned with ”Declare Function” and made good progress in developing this refactoring, but realized that we have underestimated the effort it needs . On the other hand, setting up the automated build and test server made much more problems than we expected and we therefore had a delay of several weeks early in the project, even though we started implementing ”Change Function Signature” during this time.

When the server finally was running we worked hard to regain the lost time, but the complexity of "Change Function Signature" and new problems, such as getting the C++ parser to work, made it impossible to get ourself back on the timeline.
At the end, we were glad we could finish our main feature. The three last weeks were very intensive to get the documentation finished and we would have preferred having some more time to further improve the quality of our code and our documentation.

B.3. Time Spent

As expected in every project, the time spent per week increased the closer the end got. At the beginning, we spent the planned hours in school working on the thesis. But it was no secret to us, that at the end we would have to work the after hours at home too. In the last week, we did no coding anymore, except some minor refactorings. This was mostly renaming some methods and parameters. Our official code freeze was one week prior to the dead line.

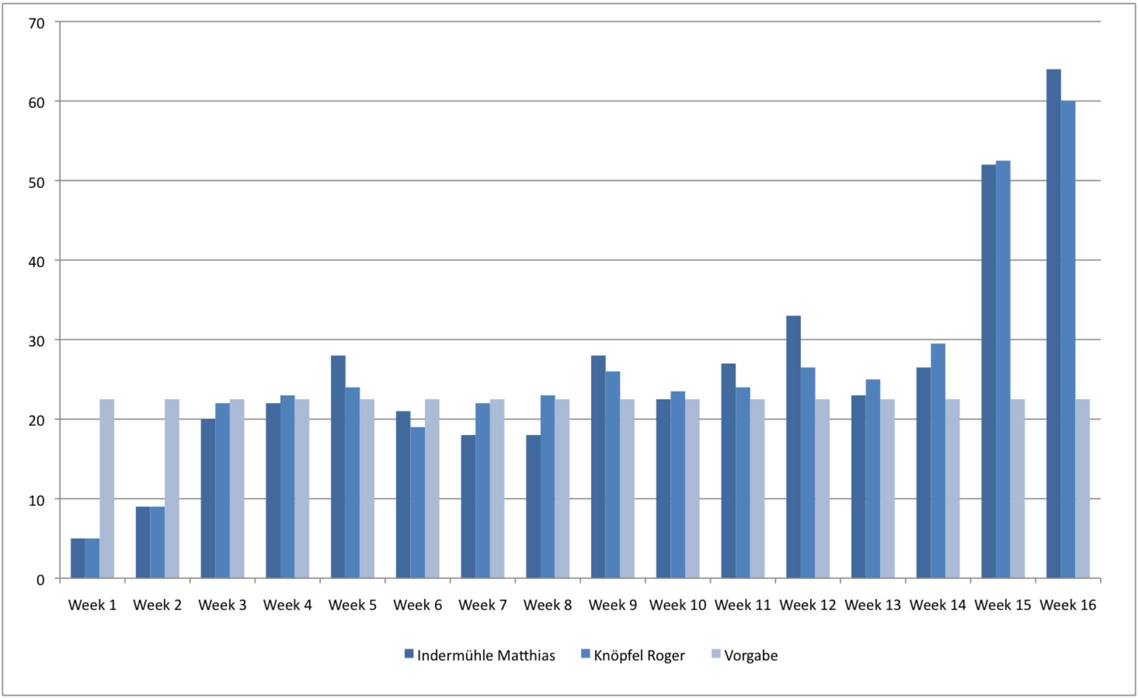


Figure B.2.: Weekly timeload

As expected, we spend about half the time implementing. Less time was planned for the preparation, but the problems concerning the build process consumed a lot of time. We tried to start with the documentation early, but as it is common with such student projects, the good intentions were abandoned somewhere in the process and the documentation work was mostly done in the last weeks.

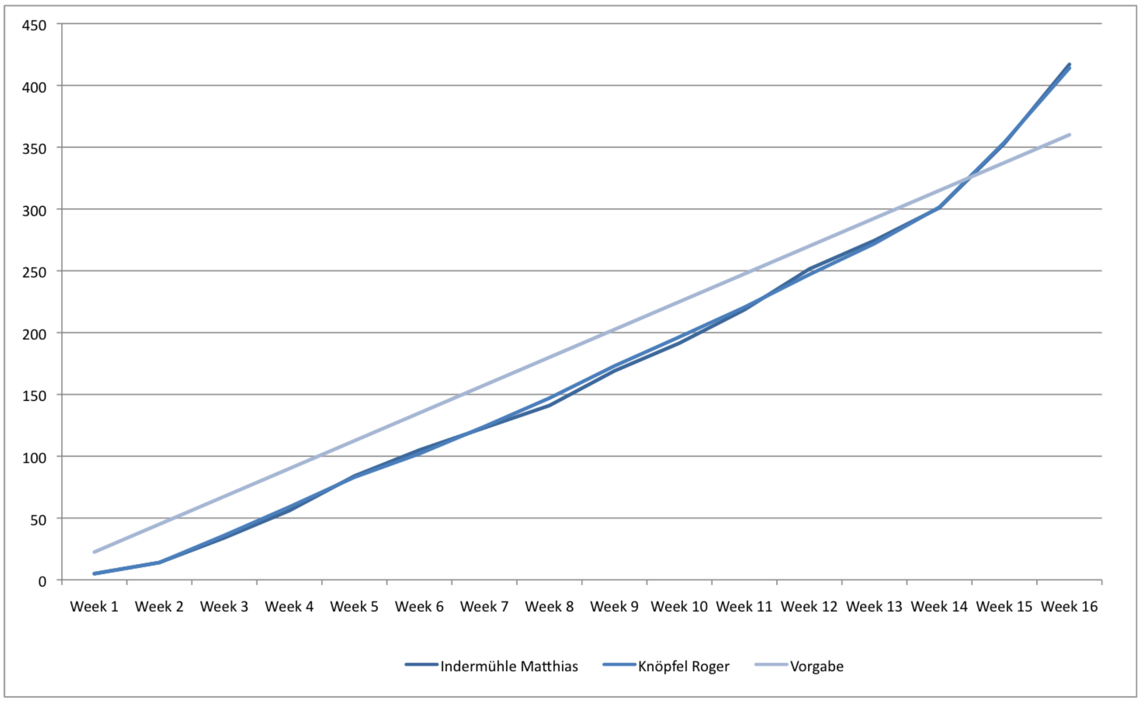


Figure B.3.: Working hours as sum over time

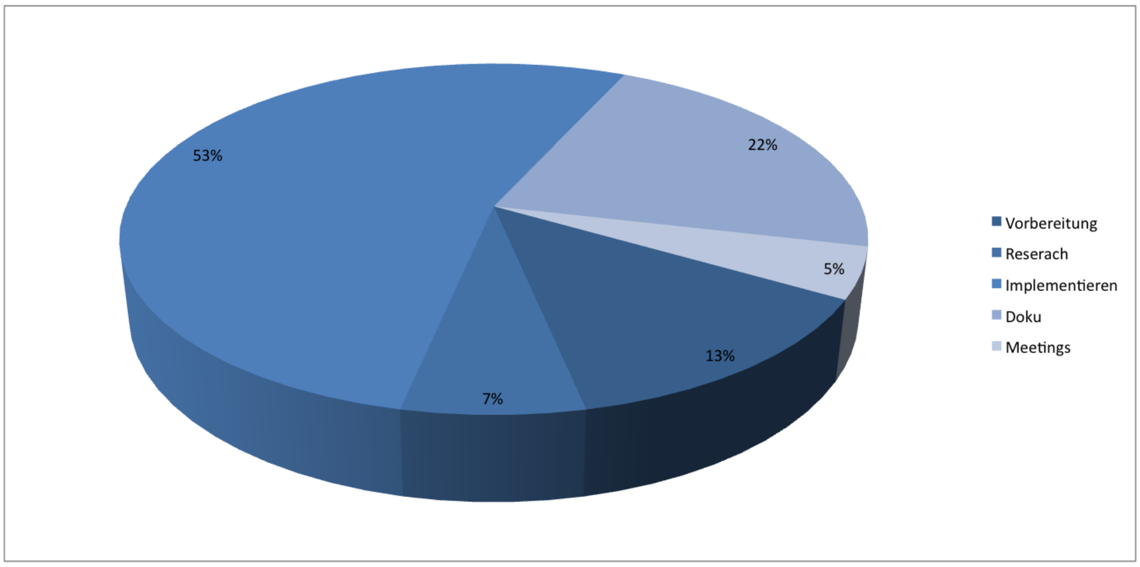


Figure B.4.: Working hours distributed over working packages

B.4. Personal Review

B.4.1. Matthias Indermuehle

For this thesis I knew what to expect, since we have already made an other refactoring for our semester thesis. As before, our first problem was the build server. Or more precisely the build process of an Eclipse refactoring plug-in. We started to develop the "Declare Function" refactoring without the Hudson server. After some time, we got the server running and the advantages of an automated built environment where astounding. I am not planning to do another software development project without one.
"Declare Function" was planned as a short introduction to get into the framework again. It turned out to be much more complex, because of the possibilities C++ offers. That was the most frustrating part about the whole work. C++ is a very complex language and there was always one more case we had to cover.
"Change Function Signature" was the main feature to do. At first, we analyzed the problem and wrote a requirements document. In retrospective, we should have spent more time on that. We had a lot of problems during the implementation, because we assumed something and forgot some cases. A lot of things we could fix again, others we had to drop. We clearly underestimated the importance of the analyze and I hope I will spent more time and effort doing research on the next piece of software I will write.
The team worked out very well, even though we regret the absence of Andrea. We never had problems to assign task and split work. Very helpful was the introduction of task issues on Redmine to coordinate our work. By creating a bug report nothing was missed or got lost.
All together, I have to say I am glad that the thesis is coming to an end, but without any doubt, I would do the same again.

B.4.2. Roger Knoepfel

After developing "Introduce PImpl" as semester thesis, which was over all a very positive experience, continuing creating a more advanced refactoring plug-in was a logical consequence. It was a pity Andrea Berweger decided not to join us, as he is a very skilled programmer, knows much about designing the GUI in SWT and was a very good and reliable partner.
The planning was a bit chaotic from the beginning, as we were not sure how to define our project goals. The only thing that was clear at first, was that we wanted to add more refactorings and that "Change Function Signature" would be our main feature. After some discussions, we further decided to develop the relatively simple "Declare Function" to refresh our skills and get a solid developing environment running. The plan was to implement more refactorings as time would permit it.
As we started working, it soon turned out that it was not programming the refactorings themselves was our main problem but getting the automated build process to work. As the tools we used were poorly documented we had to inspect other implementations to figure out how to designed the build scripts. Fortunately, Lukas Felber provided some help and it would have cost us much more time without him. Thank you very much at this place.
It soon got clear that we would not be able to implement more than the two refactorings. The diagrams of our spent time show how we had to intensify our efforts in the end the

get those two working at last.
As I look back, realize how important and yet difficult it is to make a precise project planning, especially when developing in a big framework. I will try to learn my lesson from this experience and spend more time on careful planning in the future.

C. The Project server

Software development is not any longer a task some guy does during long nights. Today it has become a team effort with the need to communicate and organize the work. Therefore some sort of collaboration platform has to be established. To do so we, we have chosen three components:

- Redmine (C.1) - A project management platform offering a wiki and a bug tracking system
- SVN (C.2) - A versioning system to manage our source code
- Hudson (C.3) - An automated build and test server

For this bachelor thesis we did not have to setup our server because we still had the old one from our semester thesis. Some updates of the software had to be done nevertheless.

C.1. Project management platform - Redmine

Redmine is an open source, web-based project management and bug-tracking tool. We used the wiki to store research information and meeting protocols, as well as the bug-tracker to plan our future tasks and manage the time we spent on them.

C.2. Versioning - Subversion

Subversion (SVN) is a versioning system. We used it to store our source code and handle the code merging when we worked on the same file simultaneously.

C.3. The build server - Hudson

We used an automated build system. It checks our code repository in intervals of ten minutes for changes. If they are any, it builds a new version of the projects and subsequently runs their attached tests. This helps us making sure our code base is stable and detecting any problems we might have overseen.

C.3.1. Tools

- Operating system: Ubuntu 8.04 LTS
- Build server: Hudson 1.352
- Builder/Target/Testrunner: Eclipse SDK 3.5.2 with CDT 6.0.2

C.3.1.1. Installation

The installation of the hudson build server on a Ubuntu 8.04 LTS is not much of a problem. There is a package in the repositories that you can install from the shell. [?]

Listing C.1: Install Hudson on Ubuntu 8.04 LTS

```
#install the repository
wget -O /tmp/key http://hudson-ci.org/debian/hudson-ci.org.key
sudo apt-key add /tmp/key
wget -O /tmp/hudson.deb http://hudson-ci.org/latest/debian/hudson.deb
sudo dpkg --install /tmp/hudson.deb

#install Hudson
apt-get update
apt-get install hudson
```

C.3.2. The build process

The build process can be splitted up into several parts:

1. Check out the code (C.3.2.1)
2. Start the frame buffer (C.3.2.2)
3. Clearing the build directory (C.3.2.3)
4. Extract Eclipse (C.3.2.5)
5. Build the features (C.3.2.6)
6. Deploy the features to Eclipse (C.3.2.7)
7. Run the tests (C.3.2.8)
8. Publish the artifacts (C.3.2.9)
9. Finish (C.3.2.10)

This process is done for both our refactorings.

C.3.2.1. Checking out the projects

The build script of the PDE would actually have the ability to checkout projects from a repository itself using CVS. Since we are using SVN, we let Hudson update the checkout folder.
The script that builds the feature is placed inside an own project which is directly checked out to the build directory.

C.3.2.2. Starting virtual frame buffer Xvfb

The test suite invoked at the end of the build does not work with a headless Eclipse. Since we do not actually need the graphical output, we simply fake the presence of a display with a virtual frame buffer called Xvfb. Xvfb is an X11 server that performs all graphical operations in memory instead of producing any output. It is installed with the package management system of ubuntu and launched by a script. In order to use the virtual frame buffer, the `DISPLAY` variable of Ubuntu has to be set. This prevented us from let Hudson start the ant script directly. We had to write a small bash script which first starts Xvfb, sets the `DISPLAY` variable and finally launches the build and test script.

Listing C.2: buld.sh - bashscript to start the whole buildprocess

```bash
#!/bin/bash

echo "BuildScript"

Xvfb -ac &
export DISPLAY=:0

ant -f build/changefunction-build.xml

killall Xvfb
```

C.3.2.3. Clearing the build directory

To get a clean workspace, all remains of the previous build are deleted.

C.3.2.4. Get the projects into position

As the build directory is now empty, we move the checked out source code of the plug-ins and the feature to the *results* directory.

C.3.2.5. Extract Eclipse

We have a specially prepared Eclipse placed on the server, which will perform the next two major steps of this build process. It is extracted to the *results* directory.
The following steps will be performed by Eclipses own ant runner, since we need its environment properties.

C.3.2.6. Build the feature

Everything is prepared to start the actual build. It is controlled by a build script provided by the PDE plug-in located in our Eclipse. The build is configured in the file *build.properties*.
The script will build all plug-ins required for the feature. At the end we get an archive containing the entire feature in a prepared folder structure.

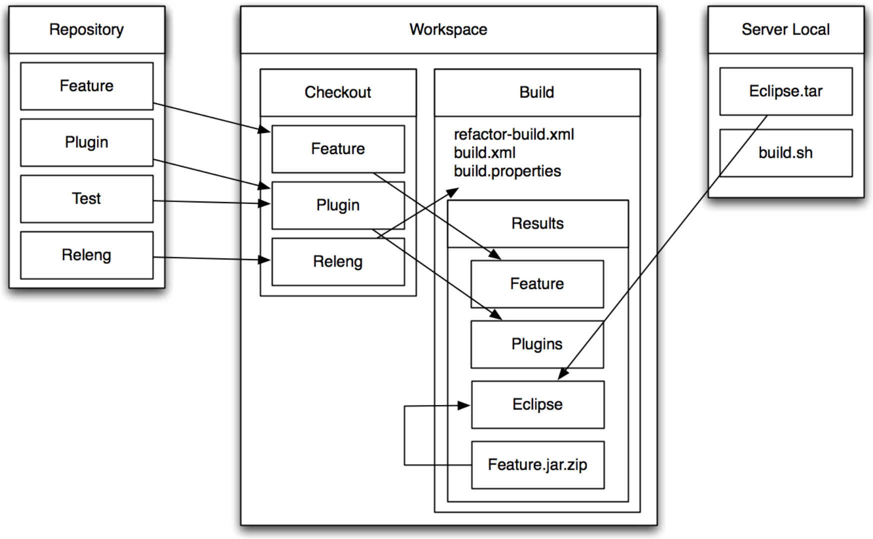


Figure C.1.: Structure of the build server's filesystem

C.3.2.7. Deploy the feature

The freshly builded feature is extracted into the Eclipse folder. Since the archive matches Eclipses folder structure, all parts are automatically at the right spot.

C.3.2.8. Run tests

The plug-in and its according tests are ready to run. The execution of the test suite is controlled by an ant script of the Eclipse test plug-in. The results of the tests are automatically parsed to a XML file.

C.3.2.9. Publish the artifacts

The result of the entire process is a ZIP file containing our refactoring plug-in. It is moved to a folder accessible from the web and a linked from Hudson's front-end. Additionally, the XML file containing the test results is read out and also published.

C.3.2.10. Finish

At last, the virtual frame buffer is stopped and Hudson collects the test results to be presented at its frontend.

Bibliography

[Beck02]	Test Driven Development: By Example, K. Beck, Addison-Wesley, 2002
[Fowler99]	Refactoring: Improving the Design of Existing Code, M. Fowler, Addison-Wesley, 1999
[Frenzel05]	The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs, L. Frenzel, 2005 Website: http://www.eclipse.org/articles/Article-LTK/ltk.html, 2010-06-11
[GrafButtiker06]	Diplomarbeit: C++ Refactoring Support fuer Eclipse CDT, Emanuel Graf, Leo Buettiker, HSR Rapperswil, 2006
[ISO14882]	ISO/IEC 14882:2003(E): Programming Languages - C++
[PImpl09]	Introduce Pimpl Refactoring, A. Berweger, M. Indermuehle, R. Knoepfel, HSR Rapperswil Website: http://sinv-56012.edu.hsr.ch/redmine/, 2009-12-16
[SWT]	Standard Widget Toolkit Website: http://www.eclipse.org/swt/

Glossary

Apache ant is a software tool for automating software build processes. 56

API (**Application programming interface**) is an interface implemented by a class to enable the use by other classes or programs.. 4

CDT (**C/C++ Development Tooling**(is an Integrated Development Environment for Eclipse for C and C++, http://www.eclipse.org/cdt. I, 3

GUI Graphical User Interface. 3

IDE (**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. 4

JDT (**Java Development Tools)** is an Integrated Development Environment for Eclipse for Java, http://www.eclipse.org/jdt. 3

JUnit is an unit test framework for Java http://www.junit.org . 9

PDE **Plug-in Development Environment** for Eclipse. 55

stub is a piece of code used to temporary stand in for some other programming functionality .. 14

use case is a description of a system's behavior as it responds to a request that originates from outside of that system. .. 12