HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Term Project, Department of Computer Science

AliExtor

University of Applied Science Rapperswil

Fall semester 2015
18. December 2015

Authors:	Özhan Kaya & Kevin Schmidiger
Advisor:	Prof. Sommerlad
Co Advisor:	Toni Suter
Partner:	IFS Institute for Software
Duration:	18.09.2015 - 18.12.2015
Workload:	240 Hours, 8 ECTS / Student
Link:	`http://sinv-56012.edu.hsr.ch/aliextor`

I Abstract

Type declarations in C++ can end up being very long and we usually don't appreciate long fragments of code. Luckily there are different ways to solve this problem by either using defines, typedefs or type alias which was introduced in C++11. Also with type alias you are able to work with templates which wasn't the case with typedefs. But like with any type of refactoring automatic is always better than manual.
The solution we provide is a refactoring plugin which can handle various amount of refactoring tasks into a type alias. Such refactoring tasks involve working with various kinds of selections which can be very tricky, especially with added options like allowing the user to refactor several occurrences so on. Also we wanted to take full advantage of the C++11 features and implemented refactoring into an alias template as well, where the user can choose which types should be parameterized.
Our final work involves a documentation (including an analysis as well as users guide), 100+ test cases, a manual on how the project setup for developing a plugin for Cevelop works and of course the working implementation of our plugin.

II Management summary

In the following, we explain the motivation and the goals of the project along with the results and possible future work.

II.1 Initial situation

Cevelop [Cev15] is an IDE (integrated development environment) for C++ programming. It initially inherits from the Eclipse CDT (C/C++ Development Tooling [CDT15]) but still lacks some features which could make programming with C++ easier. This is where our term project comes in. The goal of our plugin is to make programming in C++ with Cevelop easier by adding refactoring additions to it, which should result in automizing some code refactoring tasks, which would initially have to be done manually. Manual refactoring of software can also lead to human caused errors which can be avoided with our solution. There are various similar projects (see student project proposals site from Prof. Sommerlad [Som15a]) which all have the initial goal to make working with Cevelop easier for developers.

We liked the idea of getting into plugin development for an IDE and were generally open to new technologies, as well as working with a new infrastructure which for us also involves plugin development for an IDE.
The goals were pretty much defined open, which means, we had a few base goals like being able to create a type alias with basic node selection and the final task description (see section 4 on page 4) had to be elaborated by us until week 4. For a more technically detailed description of our function scope, see section 7.1 on page 48.

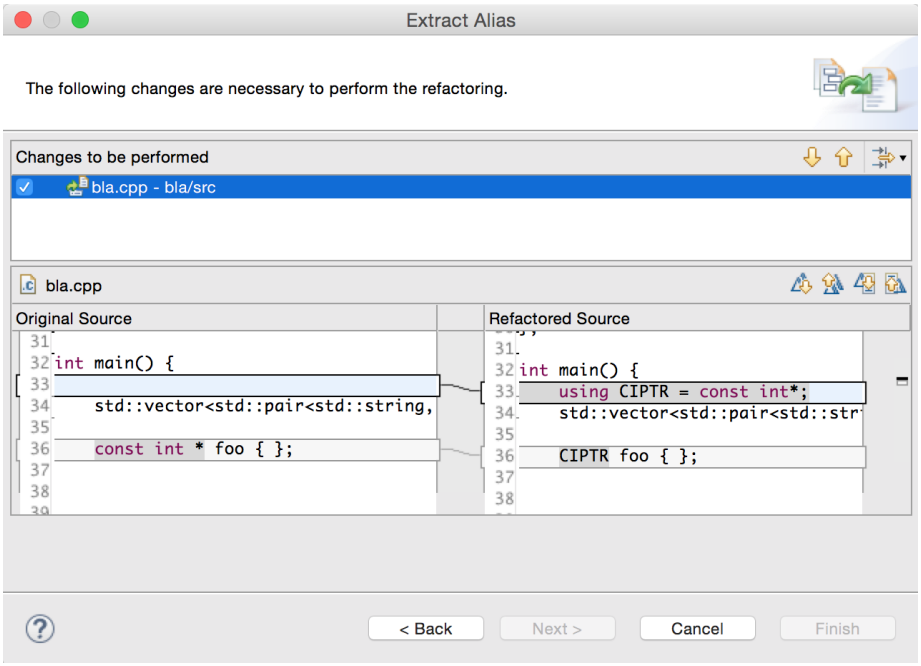

Figure 1: Simple alias extraction example

II.2 Approach

The first part of our project involved getting into the Eclipse CDT infrastructure and setting up our integration and testing server. Also by getting into plugin development in general we were better able to see how a refactoring in Eclipse can be implemented. This allowed us to implement a very basic but fully functional alias extraction which further helped us to specify our task description. After having a working prototype, we focused on our analysis which involved all the cases where the implementation of an alias extraction can be useful, as well as setting up our testing environment and getting it up and running so we can continue developing new functionality while also being able to specify test cases using the cdt-testing framework developed by the IFS (Institute for Software). While developing extraction functionality for more and more cases we quickly realized that we won't get far with our prototype code and decided to build an architecture for the refactoring cases. This allowed us to have a separation of concerns and also made further development of new wizards, visitors, refactoring cases, selection types and test cases easier. Once our architecture was done, we were able to fully focus on implementation, specifying test cases and

documenting our progress.
While developing our plugin we really appreciated the help of Toni Suter who was always very helpful when we had questions or problems which weren't always self-evident. Also we really appreciated the help of Prof. Sommerlad who assisted us while developing our task description and was always there with good input on which functionality can be added and which of those actually make sense. Our weekly meetings were a necessity since we got most of our inputs and ideas during those. We also had the ability to walk into the IFS whenever we had a question or needed an input on anything, which isn't for granted which we were also very grateful for.

II.3 Results

We managed to implement a fully functional plugin for the Cevelop IDE which is able to perform alias extraction on various types of node selections. Also we added several wizard pages with which the user can have an impact on how the refactoring should be done (e.g. choosing the type alias name, choosing if several occurrences of selected node should be refactored, how the type alias output should look like for functions, which arguments of a template should be refactored,...). We also wrote more than 100 test cases each of which specifies a different refactoring case with different selection areas of which we are especially proud. It turned out an alias extraction can be more complex than one would think. Especially with the additional goals we set ourselves, like being able to detect partial selection of a node and finding several occurrences of said selected part of the node things became quite tricky.
At first we thought the Eclipse CDT can provide us a lot of help which turned out to not always be the case. While developing our prototype, which only involved an exchange of nodes, we were able to do the biggest part of our implementation with great help from the CDT but when it came to finding several occurrences of a node, partial selection of a node or working with template arguments the CDT was more or less useless for the core functionality. This forced us to implement a lot of helper methods and business logic in general to being able to fulfill our requirements / tasks (e.g. detecting the selection range of a node). The CDT offers a lot of help when working with nodes in general but for some reason, does not have a working equality method of 2 nodes which is needed in many cases. Also when working with a partial selection we had to split a node into many parts and work with them in separate ways where the CDT provides no help.
We would've also liked to implement a refactoring of a type alias into the header file of a class or struct but due to time limitations this functionality wasn't added.

Also we underestimated the refactoring of alias template but still managed to fully implement this functionality.
While working on this term project we learned a lot about the abstract syntax tree as well as the Eclipse CDT itself. Luckily we also had some challenging tasks here and there which involved hours of brainstorming and elaboration before we could even start with implementation, which made the project overall less monotonous. Also it was nice to not create a plain CRUD (create, read, update, delete) application for once.

II.4 Forecast

Of course like with any other project there are still features which can be added to our alias extractor plugin (see section 7.2 on page 48). But we are happy with the functionality we were able to implement in the given timeframe, considering that we both had no experience on how an Eclipse plugin is built up beforehand. Also we're proud of our test cases. But unlike unit testing we can't brag with a test coverage or something alike since the possibilities with C++ syntax combined with the possible type of selection areas is enormous.

III Declaration of Authorship

We declare that this bachelor thesis and the work presented in it was done by ourselves and without any assistance, except what was agreed with the supervisor. All consulted sources are clearly mentioned and cited correctly. No copyright-protected materials are unauthorizedly used in this work.

Place and date	Özhan Kaya
Place and date	Kevin Schmidiger

Contents

4 Task description

In this section we will describe our task description for the term project. It also defines, what our goals are which were elaborated from us by week 4.

4.1 Problem

Type declarations in C++ can end up being very long and we usually don't appreciate long fragments of code. Also we often have repetitive code, which can easily be replaced with a type alias. Another benefit of a type alias is, that it also makes the code more readable. See listing 1.

```
#include <iostream>
#include <sstream> // istringstream
#include <iterator>

std::istreambuf_iterator<char> begin{in};
std::istreambuf_iterator<char> end{};
```

Listing 1: Long unreadable type

I think we can all agree that this piece of code isn't very readable. In earlier C/C++ standards there were only defines and typedefs to solve this problem. A define is basically just a textual substitution which comes with a lot of issues. The idea itself can produce a lot of ugly code and workarounds. Later on, typedef was introduced, which is no more a textual substitution but actually had a syntax and rules. With C++11 type alias was introduced, which is very similar to a typedef. However they have the advantage of working with templates. See listing 2.

```
#include <iostream>
#include <sstream> // istringstream
#include <iterator>

// using a istreambuf_iterator avoids skipping white spaces.
using Iterator = std::istreambuf_iterator<char>;

Iterator begin{in};
Iterator end{};
```

Listing 2: Type alias extracted

But often, this problem is realized, after the code is already too long for manual refactoring. Manual refactoring is not a very good option for this problem since it's

also possible to produce errors this way nor is it time efficient. Another problem is when you are working on a piece of code, which you haven't written yourself. If you want to perform a manual refactoring, you can never be sure where the same type is used as well (other files, global namespace,...).

4.2 Solution

Our solution consists of a plugin for Cevelop, which will have various options to replace declaration specifiers with a type alias. Additionally, there will be options that can automatically detect further occurrences of such declaration specifier and offer replacing of those as well. This leads to less, and more importantly cleaner code. Also you can be sure that there are no errors which could have happened in a manual refactoring.

4.3 Our goals

Since our knowledge about plugin development, refactoring and testing a refactoring is very limited, the first part of our term project will involve reading up on how to develop plugins for Eclipse CDT. Parallely, we will develop our plugin and set up our testing environment. A big part of our term project will also involve the correct manipulation of the AST (abstract syntax tree) and working with the CDT infrastructure in general. Finally we want to implement a fully functional refactoring plugin which enables such automatic refactoring with type aliases. The scope of functionality will be elaborated during the term project timespan. As an optional goal, we want to implement a type alias refactoring for an alias template.

5 Analysis

In this section we are going to analyze in which cases a refactoring with a type alias makes sense. With some code listings we will give a short introduction and also a touch on what kind of problems could occur during the refactoring and how we have to prevent or solve those problems.

5.1 General idea of type alias

In case you don't know what a type alias in C++11 is, we will briefly talk about its usage in this section. Let's start with a simple example of a type alias (listing 3).

```
int main() {
  using CIPTR = int const *;
  CIPTR const foo { };
}
```

Listing 3: Type alias

Note: This code listing is from [Som15b].

If one doesn't want to repeat int const * const foo { 42 }; over and over again and shorten it by using CIPTR instead, a type alias comes in handy. If given a name, which represents the type well, it results in more readable code. In listing 3 the type isn't very long and it would be okay writing it all out. But imagine one is implementing a class which has to handle the type in listing 4.

```
#include<vector>
#include<iterator>

class myClass {
  using crev_iterator = std::vector<const int>::const_reverse_iterator;
};
```

Listing 4: Complicate Type

It would be very frustrating typing this type over and over again and it'd probably not be easily readable as well.

5.2 The AST in Eclipse for C++

Before we get into the refactoring cases, you'll need some brief knowledge about the AST (abstract syntax tree). In Eclipse the C++ code is parsed to an AST. If refactorings are made, this AST is modified and in order of the modified AST the code is generated to the editor. In this section we analyze what kind of nodes we have to deal with and what kind of complications could occur.
The PASTA (Painless AST Analysis [IFS15a]) Plugin from IFS [IFS15b] gives a nice view of the AST. Let's show from an example in listing 5.

```
int main() {
  int a { 42 };
}
```

Listing 5: Example Code for PASTA

The AST of this code is shown in figure 2 on page 8.

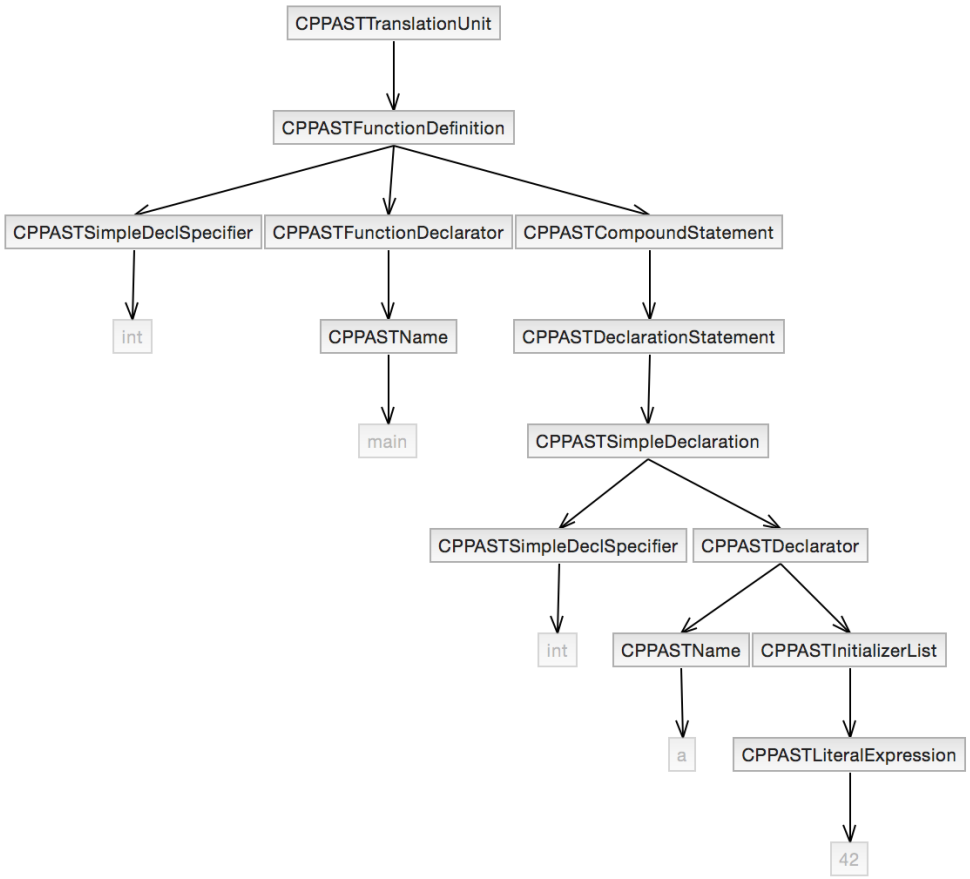

Figure 2: AST from PASTA listing 5 on page 7

5.3 Possible refactoring cases

In this section we will analyze each refactoring case and touch on if such refactoring makes sense as well as what the difficulties with each refactoring case are.

5.3.1 Declaration statement

This refactoring describes a type alias generation from a simple declaration statement. This case looks like it's very easy to implement but we encounter some difficulties very quickly. In this section, we will talk about refactoring following simple

declaration case.

```
int main() {
  int const * const foo { };
}
```

Listing 6: Simple declaration

First we want to look at which parts of this simple declaration can actually be put into a type alias (syntactically) and in which cases it actually makes sense (semantically). A type alias requires a type id. We do not want to get into too much detail since there are quite a lot of cases of type id's which can be used in a type alias. More information on type alias see [ISO14, page 151, 7.1.3 The typedef specifier] and type id's also see [ISO14, site 191, 8.1 Type names].

The cases, both syntactically and semantically, of the listing 7 make sense to be put in a type alias.

```
using alias = int;

using alias = const int;

using alias = const int *;

using alias = const int * const;
```

Listing 7: type alias of int

Extracting only a const or only a pointer is syntactically incorrect. Every type alias needs a type. But we end up with a bigger implementation related problem when we look at the listing 7 example one and three. This problem will be discussed in section 6.3.2 on page 38.

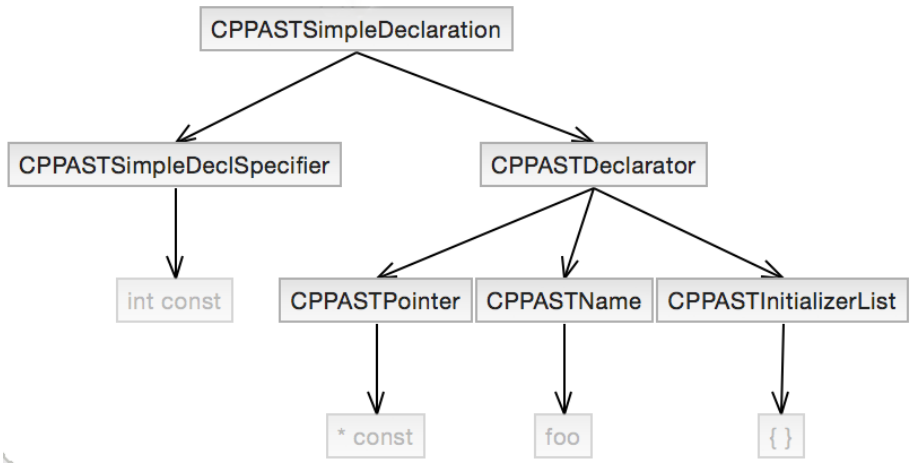

Figure 3: AST of a const int * const declaration

There are also two examples, which are syntactically correct but have another meaning when extracted.

```
using alias = int *;
const alias const foo { };
```

Listing 8: wrong usage of type alias

```
using alias = int * const;
const alias foo { };
```

Listing 9: wrong usage of type alias

The problem here is that the type (in our case, int) is boxed into a pointer which is stored in the type alias. When adding additional type qualifiers like const to said type alias, it directly references the pointer (because of the boxing). So the issue is that when we do such extraction we actually change the semantics of the existing code which isn't allowed under any circumstances in a refactoring.

Let's take a look at a bit more complex declaration statement and the issues that can arise with refactoring them.

```
std::vector<std::pair<std::string, int>> bar { };
```

Listing 10: vector of pairs of string and int's

At first glance, this declaration statement might not look much different from the first example. But if we have a look at the AST of it, we quickly realize that this is a different case. See figure 4 on page 12.

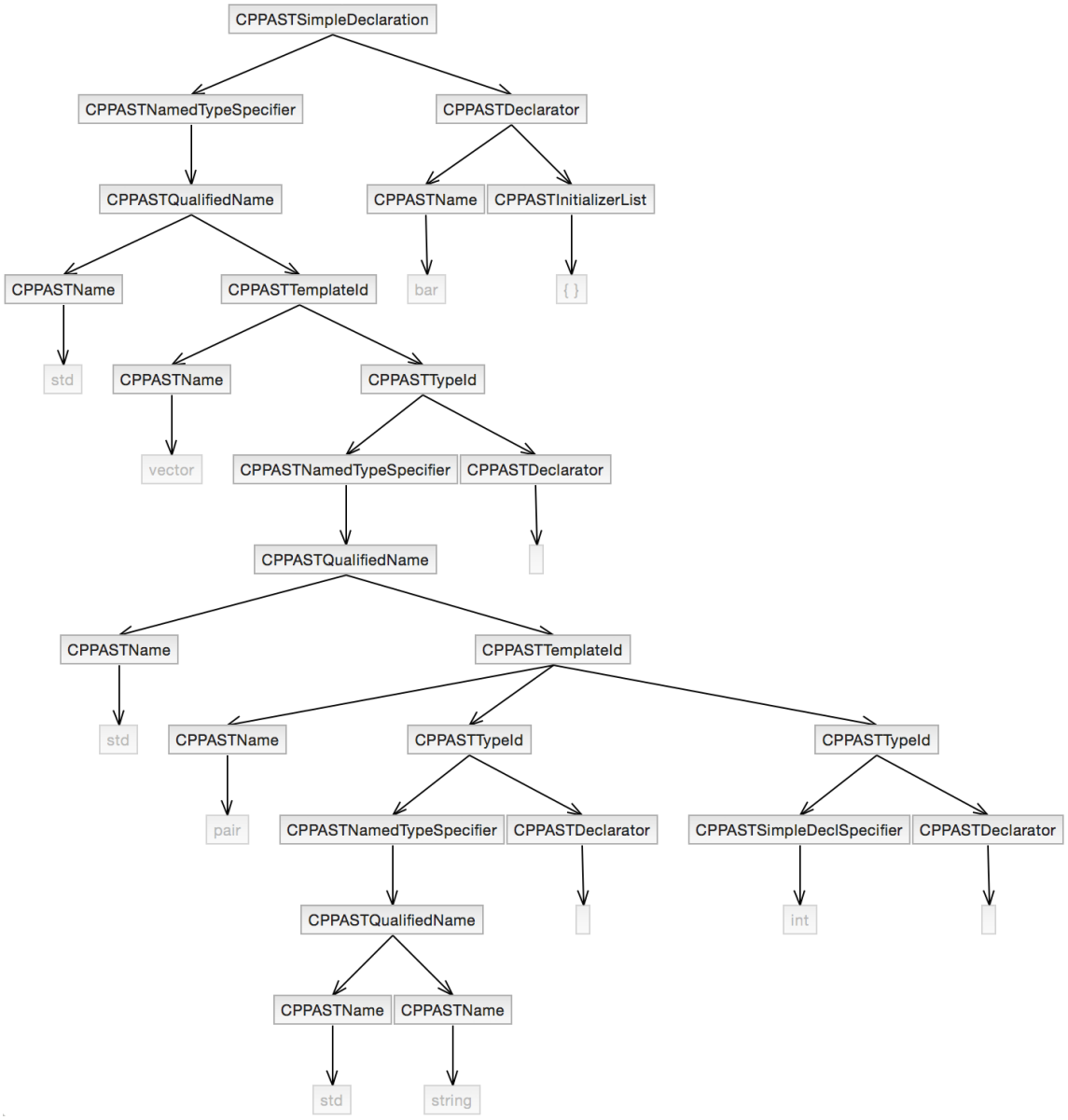

Figure 4: more complex AST example from listing 10 on page 11

As you can see, this type of declaration statement is way bigger, but overall it's still just a simple declaration. The problems start to arise once we want to make a selection (selecting the part which should be refactored). For example if we select

std::vector<std::pair<**std::string**, int>>bar{ };
depending on how the selection is implemented we could either end up with a CPPASTTypeId, CPPASTNamedTypeSpecifier or a CPPASTQualifiedName. If we want to find same occurrences of this node things start to become even worse. We could have another std::string appearance in our code which is for example a CPPASTTypeId with an empty declarator and our selected node is a CPPASTQualifiedName. Comparing these two nodes can be painful since there is no working implementation of comparing the textual content of these nodes in the Eclipse CDT. And even if you want to compare nodes with their textual content, you quickly encounter problems like "int const" and "const int" being the same semantically, but not syntactically. An own implementation is required in this case.

5.3.2 Function parameter

A function parameter in the Eclipse CDT is defined as a CPPASTParameterDeclaration which belongs to a CPPASTFunctionDeclarator which itself can have various amount of CPPASTParameterDeclarations. The only tricky part of this refactoring is that maybe we want to refactor both function declaration as well as the implemented function. The implementation has to be independent of if a parameter declaration of a function declaration or of a implemented function has been selected.

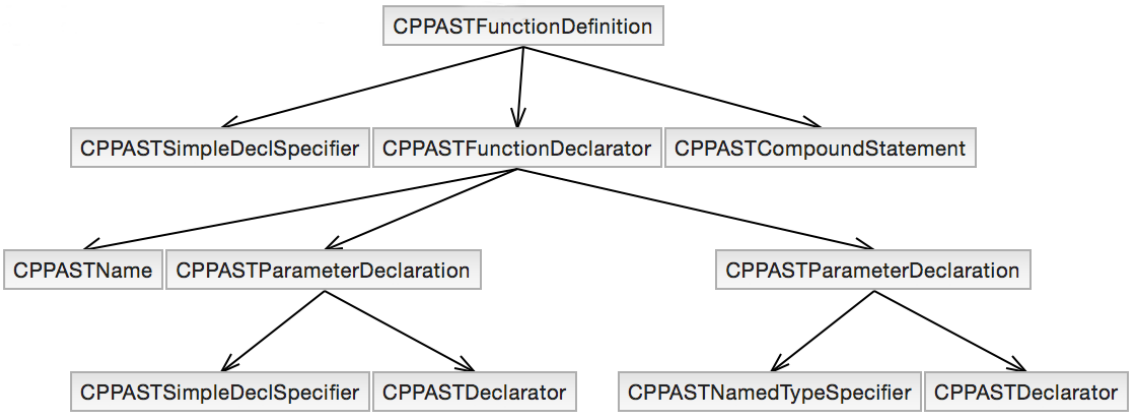

Figure 5: AST of Function Parameters

You can see that after the ICPPASTParameterDecleration there are known types

like the SimpleDeclSpecifier and the Declarator. The SimpleDeclSpecifier from the ICPPASTFunctionDefinition is the return type, which is explained in the next section.

5.3.3 Return type

Using declarations can also be used on a return type of a function declaration. This case does not have to be handled in a special way (see Figure 6 on page 15, which is a known type), but we have to keep in mind that scoping will become an issues, if we want to use a type alias in both function declaration and implementation (since the function declaration is usually found in the header and the implementation in the .cpp file). We will discuss scoping in another subject. In listing 11 is an easy example without any scoping issues included.

```
using awesomeResult = int;
awesomeResult calc(int num1, int num2, char operatorSign);
```

Listing 11: Return type example with type alias

5.3.4 Cast Expressions

When looking into a static cast, the selected node is called CPPASTCastExpression which consists of a CPPASTTypeId and a CPPASTLiteralExpression. This case can also be handled fairly easy since we already know how to deal with type aliases on TypeIds from earlier examples. So implementation shouldn't be the hardest of tasks but it's something which still has to be considered.

```
using desiredType = int;
desiredType n = static_cast<desiredType>(3.14);
```

Listing 12: Static cast with type alias

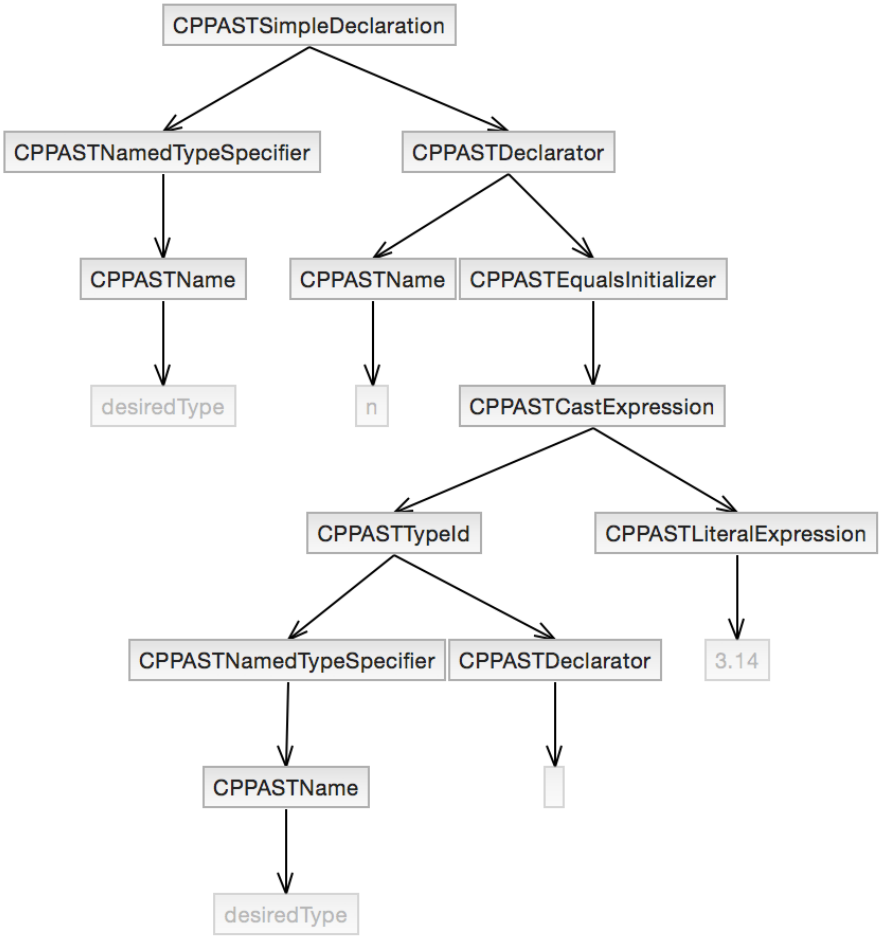

Figure 6: AST of a static cast from listing 12 on page 14

The plain old type of casting where you specify the desired type in parentheses is also defined as a CPPASTCastExpression. So this case is the same to implement. This is also the case for a dynamic_cast and a const_cast.

5.3.5 Template parameter (Alias Template)

Sometimes it is not known, what kind of type you want to use, or you want it to be more generalized. So one can use an alias with template parameter.

```
int main() {
  template<class T1, class T2>
  using myMap = std::vector<std::pair<T1, T2>>;

  myMap<int, std::string> foo {};
}
```

Listing 13: Alias with Template Parameters

This case is a bit different from what we already know. The AST of it has other nodes.

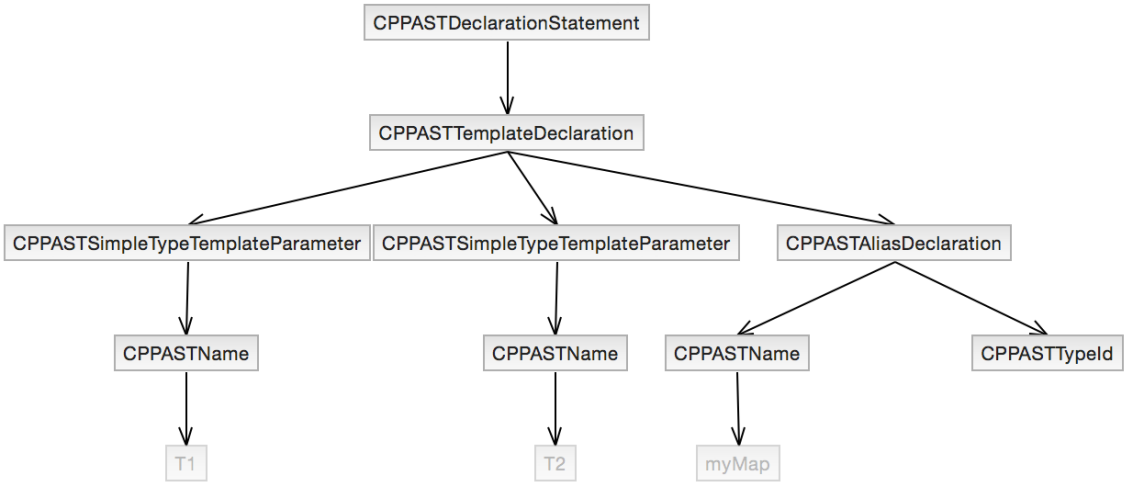

Figure 7: AST of Template Arguments from listing 13

As we can see the template parameters have their own node called ICPPASTSimpleTypeTemplateParameter, which contain a ICPPASTName and that node contains the parameter. And then together with an ICPPASTAliasDeclaration, which also contains a name and a TypeId it builds a ICPPASTTemplateDeclaration.

The problem with refactoring this type of constellation is to determine what was actually selected. And if there are template arguments [ISO14, site 331, 14.1 Template parameters], we have the user to decide if he wants to use template parameters or if he wants it hard coded. Another thing we have to check is, if the extraction is in a class or struct (in further cases just class is used, but could be both) scope. But this will be explained in the next section. It could be possible, that the extraction uses

a template parameter, which is used from the class.

```
template<class T1, class T2>
class MySpecialMap {
  std::vector<std::pair<T1, T2>> map {};
};
// after extraction
template<class T1, class T2>
class MySpecialMap {
  using myMap = std::vector<std::pair<T1, T2>>;
  myMap map {};
};
```

Listing 14: Template Parameters in a class

The problem here is we have to check if the extraction is in the class scope. A look in the AST will help here.

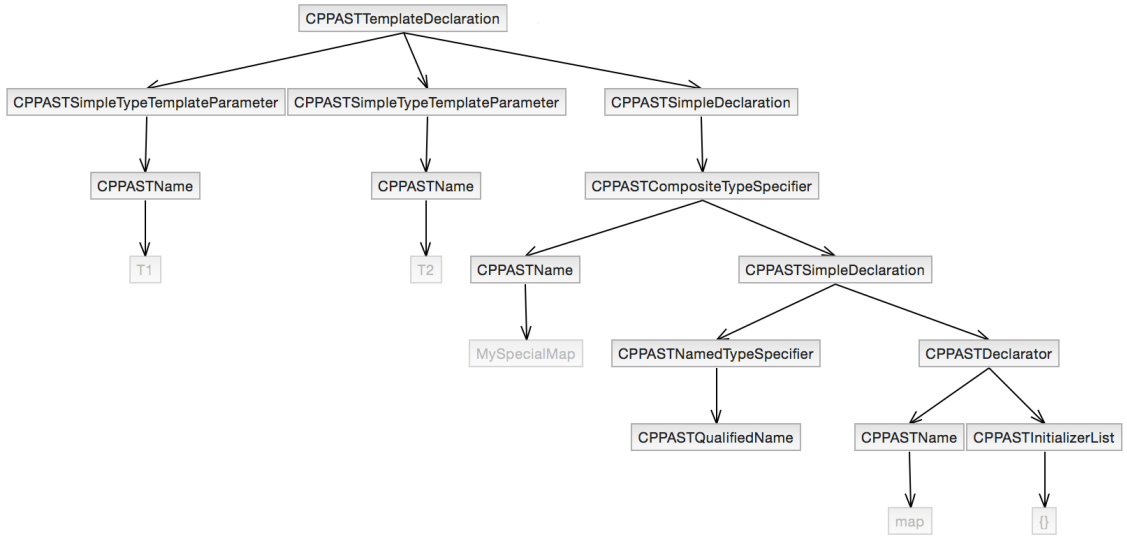

Figure 8: AST of listing 14

Note: This is the AST view from before the extraction of the code listing 14.

You can see that a class with template parameters is represented as an ICPPASTTemplateDeclaration with the parameters as ICPPASTSimpleTypeTemplateParameters and the class itself as a SimpleDeclaration or a ICPPASTCompositeTypeSpecifier. The CompositeTypeSpecifier only contains the statement, which is extracted

to myMap. So we have to evaluate if the extraction is in a CompositeTypeSpecifier and this CompositeTypeSpecifier must be in a TemplateDeclaration. Then we would have to take the parameters and use it for the extraction instead of using an own template declaration.

5.3.6 Type alias in classes or structs

While defining namespaces with your defined alias types means that they could be accessed from everywhere, if the namespace is used, sometimes it doesn't makes sense that these types could be accessed. Especially if they are declared in a class and just used there they don't have to be „exposed“. So declaring them private makes more sense.

```
// mySpecialClass.h
#ifndef MYVERYSPECIALCLASS_H_
#define MYVERYSPECIALCLASS_H_

#include <vector>
#include <initializer_list>

class mySpecialClass {
  using vector = std::vector<const int>;
  using value_type = vector::value_type;
  vector vec;
public:
  mySpecialClass(std::initializer_list<const int> list) :
      vec(list) {
  }
  value_type front() const;

  int specialfunc(int, int);
};
#endif /* MYVERYSPECIALCLASS_H_ */

// mySpecialClass.cpp
#include "myVerySpecialClass.h"

mySpecialClass::value_type mySpecialClass::front() const {
  return vec.front();
}

int mySpecialClass::specialfunc(int x, int y){
  return 42;
}
```

Listing 15: Type alias in a class

It could also be possible, that the extraction one wants to make, uses a template parameter, which is used from the class.

```
template<class T1, class T2>
class MySpecialMap {
  std::vector<std::pair<T1, T2>> map {};
};
// after extraction
template<class T1, class T2>
class MySpecialMap {
  using myMap = std::vector<std::pair<T1, T2>>;
  myMap map {};
};
```

Listing 16: Template Parameters in a class

5.3.7 Type alias in namespaces

Sometimes it makes sense to use an alias in a namespace. This would look like the listing 17.

```
#include<vector>

namespace demo {
  using myvec = std::vector<const int>;
}

demo::myvec foo() {
  return demo::myvec {42};
}

int main() {
  demo::myvec other = foo();
}
```

Listing 17: Type alias in namespaces

Declaring a type alias in a namespace means that one can use this type in another file. For example it would be possible to move the function foo() in another file.

Now our problem here is to find the other scope, which one wants to extract the alias. Or maybe the namespace has to be built first, because it doesn't exist. If the namespace already exist we would have to find the ICPPNameSpaceDefinition and place the alias inside of it. To find the namespace (hopefully in the same file) we would need the name of it (as a user input) and then we can search for it. If the namespace doesn't exist we'd have to create one first. By creating the namespace it is possible to place the alias in it and voilà, there we go with an alias in a brand new

namespace. The question is, if one wants to create a new namespace since the rule is usually as locally as possible when it comes to variables.

5.3.8 Function declaration

If one has to deal with a lot of functions in his code and lots of them are having the same signature except for the name, it is easier to use an alias of this function declaration. An alias in this case makes the code also shorter and easier readable, if there are good names used for the alias.

```
// before
int add(int x, int y){
  return x + y;
}

int calc(int x, int y, int (*calcFunc)(int,int)){
  return calcFunc(x, y);
}

int main(){
  int x { calc(1, 2, add) };
}
// after
using calcFunc = int (int, int);

int add(int x, int y){
  return x + y;
}

int calc(int x, int y, calcFunc f){
  return f(x, y);
}

int main(){
  int x { calc(1, 2, add) };
}
```

Listing 18: Type alias for functions as Parameter

Sometimes it is useful to use an alias for functions. This alias could be a normal function definition or a function pointer or a function reference as shown in listing 19.

```
using func = void (int, int);
using fptr = void (*) (int,int);
using fref = void (&) (int, int);

// Forward declaration
void example(int, int);
```

```
int main() {
  fptr fn = example;
  fn(3, 4);
}
```

Listing 19: Type alias for functions

Note: This code listing is from cppreference [Cpp15] but it is extended with the function pointer and reference.

5.4 Scoping

A type alias doesn't have to be in the same scope as where it is used, but it could be used in a subscope of it.

```
#include<vector>

using myvec = std::vector<const int>;

myvec foo() {
  return myvec { 42 };
}

int main() {
  myvec other = foo();
}
```

Listing 20: Type alias in subscopes

Note: We would not recommend to use type alias in such way, because global variables are evil.

But as you can see a type alias can be used in various subscopes. This leads to an even better way using a type alias.

5.4.1 Extraction Scope

Mostly alias extraction are used or needed in the same scope. But as seen in section 5.4 on page 21 one maybe wants to replace a type with an alias in other scope. This means, we have to scan the file and find other occurrences of the type. But

maybe it is not needed to replace all of this types to an alias. This leads us to provide a refactoring wizard, which gives options on which type should be replaced or not.

If the extraction is in the same scope, it is also necessary to place the alias at the beginning of the scope. Otherwise the alias wouldn't be known and we'd only produce a compile error.

5.4.2 Extraction in structs or classes

Building classes involves handling with several types. But it is not always needed to „expose“ them, so they are made private. So the tricky part is the following. If one is building up a class and sees that a type could be used more often, it would be nice to extract it to an alias. For example the code of listing 21 is given.

```
/*---------Header---------*/
#ifndef AUTO_H_
#define AUTO_H_

struct Auto {
  int speedUp(int);
};

#endif /* AUTO_H_ */

/*---------Class---------*/
#include "auto.h"

int Auto::speedUp(int kmh) {
  kmh++;
  return kmh;
}
```

Listing 21: Code before

And the code wanted in listing 22.

```
/*---------Header---------*/
#ifndef AUTO_H_
#define AUTO_H_

struct Auto {
  using speed = int;
  speed speedUp(int);
};

#endif /* AUTO_H_ */
```

```
/*---------Class---------*/
#include "auto.h"

Auto::speed Auto::speedUp(int kmh) {
  kmh++;
  return kmh;
}
```

Listing 22: Desired code

One problem for us is, the scope. As seen in the code listing, the type alias doesn't have to be in the same scope or file. So we have to refactor it from the cpp file to the header file. Another thing we have to consider is, if this alias should be private or not. So we have to find the right place to place the alias. If no private space exists and we're working with a struct, we'd have to create one first.

6 Implementation

6.1 Eclipse Plugin

While developing our Eclipse plugin, we had to adhere to a given infrastructure. In this section we will briefly touch on how a plugin works without going too much into the details. If you want to read up on a more detailed description on how a plugin has to be / can be implemented you may have a look at the official description [Ecl15a] or check out the master-thesis TurboMove [Yve11] from Yves Thrier.

6.1.1 Our additions

Our plugin extends two areas of the Eclipse UI. One is an additional entry in the refactoring menu at the top while the other is a refactoring menu action entry for the C/C++ Editor context. These entries will be coupled with a unified program logic. Depending on what type of text selection is made, several options will be presented to the user through a wizard.

6.1.2 Activator

The activator is the entry point of your Eclipse plugin. It can be used to control the lifecycle of a plugin. Basically the activator is just generated boilerplate code. What we did additionally to build up on it is a logger which we can use from any part of our plugin. This logger will probably not find much use in the final product but can be quite handy during development.

6.1.3 Extension Points

Extension points are points in Eclipse where a new plugin can connect into. Using these extension points, you can add additional menu entries and a lot more into many of the already existing UI areas of Eclipse.
For this term project we only wanted to add additional entries in the refactoring menu tab as well as the refactoring popup menu inside the editor. The refactoring menu tab belongs to the JDT (Java development tools). The refactoring popup menu inside the editor is an extension point from the Eclipse CDT.

More on how we used the extension points can be found in the chapter "plugin.xml" 6.1.4.
Want to know more about Eclipse extension points? Here's a good tutorial from Vogella on Eclipse extension points [Vog15a].

6.1.4 plugin.xml

Extension points in Eclipse are added using a plugin.xml file. In the code fragments below you can see what we've added to the extension points.

```
<extension
  point="org.eclipse.ui.actionSets">
  <actionSet
    id="ch.hsr.ifs.cute.aliextor"
    label="C/C++ Coding"
    visible="true">
    <menu
      id="org.eclipse.jdt.ui.refactoring.menu"
      label="Refactor">
      <separator name="codingGroup"></separator>
    </menu>
    <action
      class="ch.hsr.ifs.cute.aliextor.ui.AliExtorRefactoringActionDelegate"
      id="ch.hsr.ifs.cute.aliextor.menutab"
      label="Extract Alias..."
      menubarPath="org.eclipse.jdt.ui.refactoring.menu/codingGroup">
    </action>
  </actionSet>
</extension>
```

Listing 23: JDT Refactoring Menu Tab Extension

```
<extension
  point="org.eclipse.ui.popupMenus">
  <viewerContribution
    id="ch.hsr.ifs.cute.aliextor.viewerContribution"
    targetID="#CEditorContext">
    <action
      id="AliExtor_HelloWorld.popup"
      label="Extract Alias..."
      class="ch.hsr.ifs.cute.aliextor.ui.AliExtorRefactoringActionDelegate"
      menubarPath="org.eclipse.cdt.ui.refactoring.menu/codingGroup">
    </action>
  </viewerContribution>
</extension>
```

Listing 24: CDT Refactoring Popup Menu Extension

Note: The full plugin.xml file is located directly inside the main plugin folder.

There are actually two ways to build up on the extension points. Sadly we had to use a deprecated way with the actionSets due to Eclipse CDT only being able to work with those.

6.1.5 UI logic

The UI logic for an Eclipse plugin is given and does not leave much room for own innovation. More on what had to be done to get our refactoring entries inside the Eclipse UI working will be discussed in 6.2.1.5.

6.2 Overview

In this section we'll give you a brief overview of our implementation. We will mainly focus on the packaging and abstraction side of things.

6.2.1 Packages

Our plugin is separated into six main packages. In this section we will briefly look into each package and explain its belongings.

6.2.1.1 ch.hsr.ifs.cute.aliextor

This package only includes the activator which we discussed in chapter 6.1.2 on page 24. The idea was to store a global collection of nodes to make refactoring easier. But we later decided that it makes more sense to store the found nodes in each visitor.

6.2.1.2 ch.hsr.ifs.cute.aliextor.ast

This package includes everything that's related to the AST. Visitors which are used to traverse through the AST will be implemented here.
We don't want to get into too much detail here since the visitors themselves are pretty much self explanatory. Depending on what type of refactoring we want to perform, we use different visitors. The main job of the visitors are to find nodes and

check if they are equal to the selected node and add them to an occurrence list. This also includes several equality checks which differ from node to node (especially when we come to partial node selection).
Here's a list of which type of nodes we usually work with to get a better understanding about which nodes our plugin operates on:

- DeclSpecifiers
- Names
- TypeIds
- Declarations
- ParameterDeclarations

We also had to operate on different type of nodes such as NamedTypeSpecifier, FunctionDeclaration, AliasDeclaration, CompositeTypeSpecifier and many more which we do not explicitly visit.

6.2.1.3 ch.hsr.ifs.cute.aliextor.ast.selection

All selections inherit from an interface called IRefactorSelection which only has one method called getSelectedNode(). We have chosen this approach since we were not sure what type of selections we'd have to deal with in the future. It turned out that we only have to differentiate between a normal node selection and a partial selection (for example: selecting only "int" from a "const int"). More on each selection type will be explained in 6.3.1.1 on page 36 and 6.3.2.1 on page 39.

6.2.1.4 ch.hsr.ifs.cute.aliextor.refactoring

This package includes the main logic of our refactoring cases. A big problem we had with our architecture was that depending on a selection we had to run either refactoring X or refactoring Y. But the refactoring, which extends from CRefactoring is needed in different parts like our RefactoringRunner which we'll discuss about later and also in our BaseTest which extends from CDTTestingRefactoringTest which also requires us to implement a createRefactoring() method. While being in these two stages of the code we had no access to things like AST, NodeSelector and Selection. That's where we decided to build a bit of architecture for our refactoring cases. Our

final architecture, which is a mix of the strategy pattern as well as the proxy pattern looks like the figure 9.

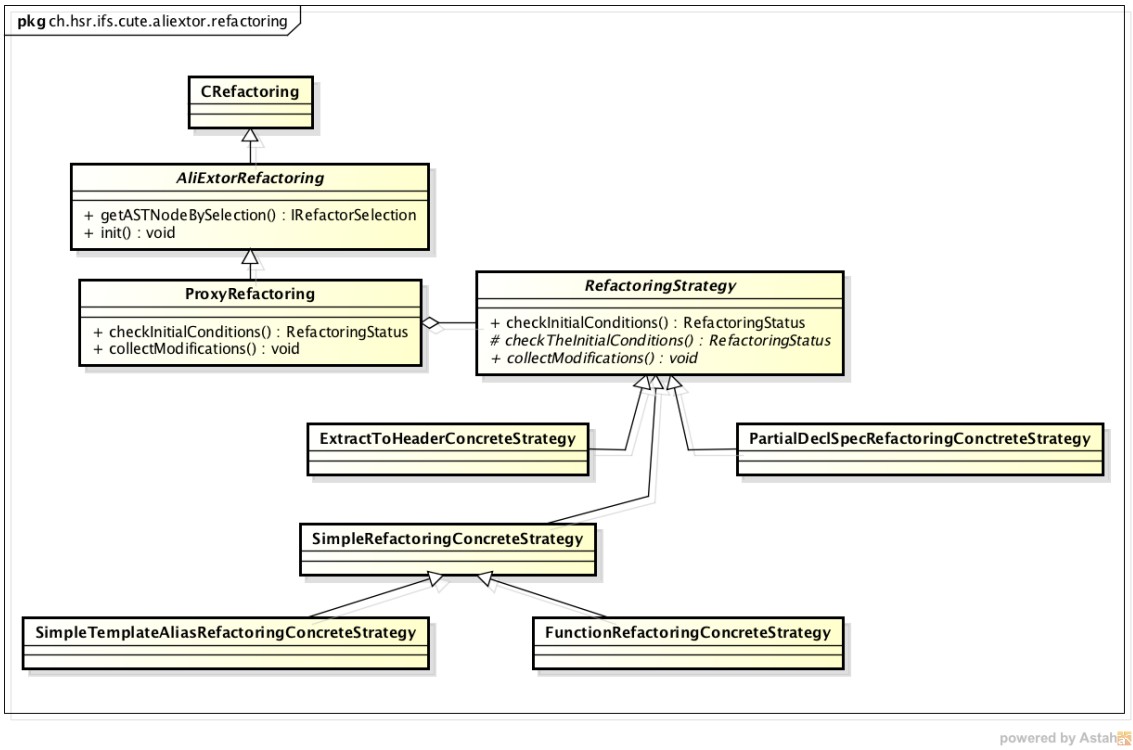

Figure 9: UML of refactoring architecture

One could argue if merging the AliExtorRefactoring with ProxyRefactoring makes sense but we liked the idea of having our proxy logic separated from the rest. Depending on what selection was detected by our getASTNodeBySelection() method we end up with a different type of selection which leads to our proxy calling a different strategy.

6.2.1.5 ch.hsr.ifs.cute.aliextor.ui

This package holds the UI logic elements (delegate, action, runner).

ExtractAliasRefactoringActionDelegate
Since it is possible to execute our refactoring through the refactoring menu in the toolbar as well as through the in editor popup we will have to link them to a single implementation (duplicate code is evil). This is where the ExtractAliasRefactoringActionDelegate comes in handy. This class implements two interfaces.
The first interface called IWorkbenchWindowActionDelegate is used to contribute to Eclipse through the menu toolbar. This implements a method called init() which you can use to get the active window (needed for further steps). The method run() is what's called when an actual action has to be executed. Since the window has been initialized in the init() method, we're fine to go.

```
@Override
  public void init(IWorkbenchWindow window) {
    this.window = window;
  }
```

Listing 25: init() from IWorkbenchWindowActionDelegate

Of course the user can also access the ExtractAliasRefactoringActionDelegate through the in editor popup. That's what the interface IEditorActionDelegate can be used for. It introduces a method called setActiveEditor() which can also be used to get the window in which we want the refactoring to be done. Also it implements the run() method discussed earlier.

```
@Override
  public void setActiveEditor(IAction action, IEditorPart targetEditor) {
    if (targetEditor != null) {
      window = targetEditor.getSite().getWorkbenchWindow();
    }
  }
```

Listing 26: setActiveEditor() from IEditorActionDelegate

Finally we need a way to tell both buttons that ExtractAliasRefactoringActionDelegate is their shared logic. This can easily be done with the plugin.xml file (see 6.1.4 on page 25). Both extension point entries have a common class (AliExtorRefactoringActionDelegate) declared in their action.

ExtractAliasRefactoringAction
From this point on, it doesn't matter anymore from which UI element we reached this piece of code. We extend from RefactoringAction which forces us to overwrite

a run() method. From the run() method we call our runner. Aside from that the action class does not implement any further logic.

ExtractAliasRefactoringRunner
Finally we reached our runner. This is where the wizard and the refactoring itself will be called. We extend from RefactoringRunner which gives us another run() method. We create an instance of our ProxyRefactoring as well as our Wizard and call a run() method using the created instances. This merges our wizard and our refactoring logic.

```
@Override
public void run() {
  AliExtorRefactoring refactoring = new ProxyRefactoring(element, selection, project
    );
  RefactoringWizard wizard = new Wizard(refactoring, RefactoringWizard.
    DIALOG_BASED_USER_INTERFACE | RefactoringWizard.WIZARD_BASED_USER_INTERFACE);
  this.run(wizard, refactoring, RefactoringSaveHelper.SAVE_REFACTORING);
}
```

Listing 27: run() from RefactoringRunner

6.2.1.6 ch.hsr.ifs.cute.aliextor.wizard

In this section we shortly describe the role of our wizard. A wizard is necessary as we need an input from the user defining the new name of our type alias (needed for any type of type alias refactoring) and depending on which refactoring we initiate we can provide additional options.
Secondly we may need to know where to place our type alias, but often it is just used in the same scope. Also we need to know if we should replace all occurrences with the alias or just the selected node. This is where our wizard comes in handy.
In the Figures 10 to figure 13 on the following pages are some images which help illustrate which options we actually have implemented in our final product.

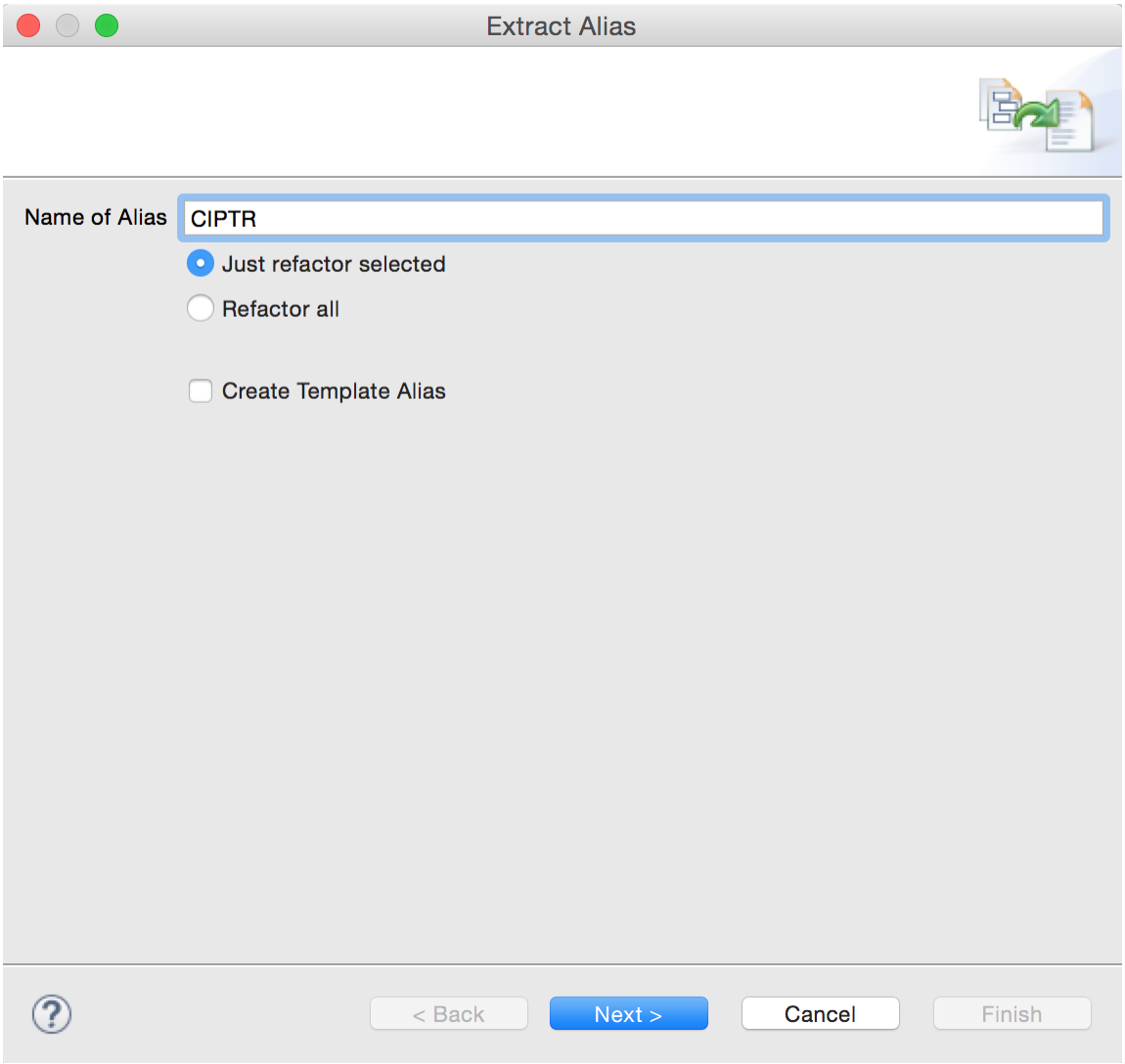

Figure 10: standard wizard

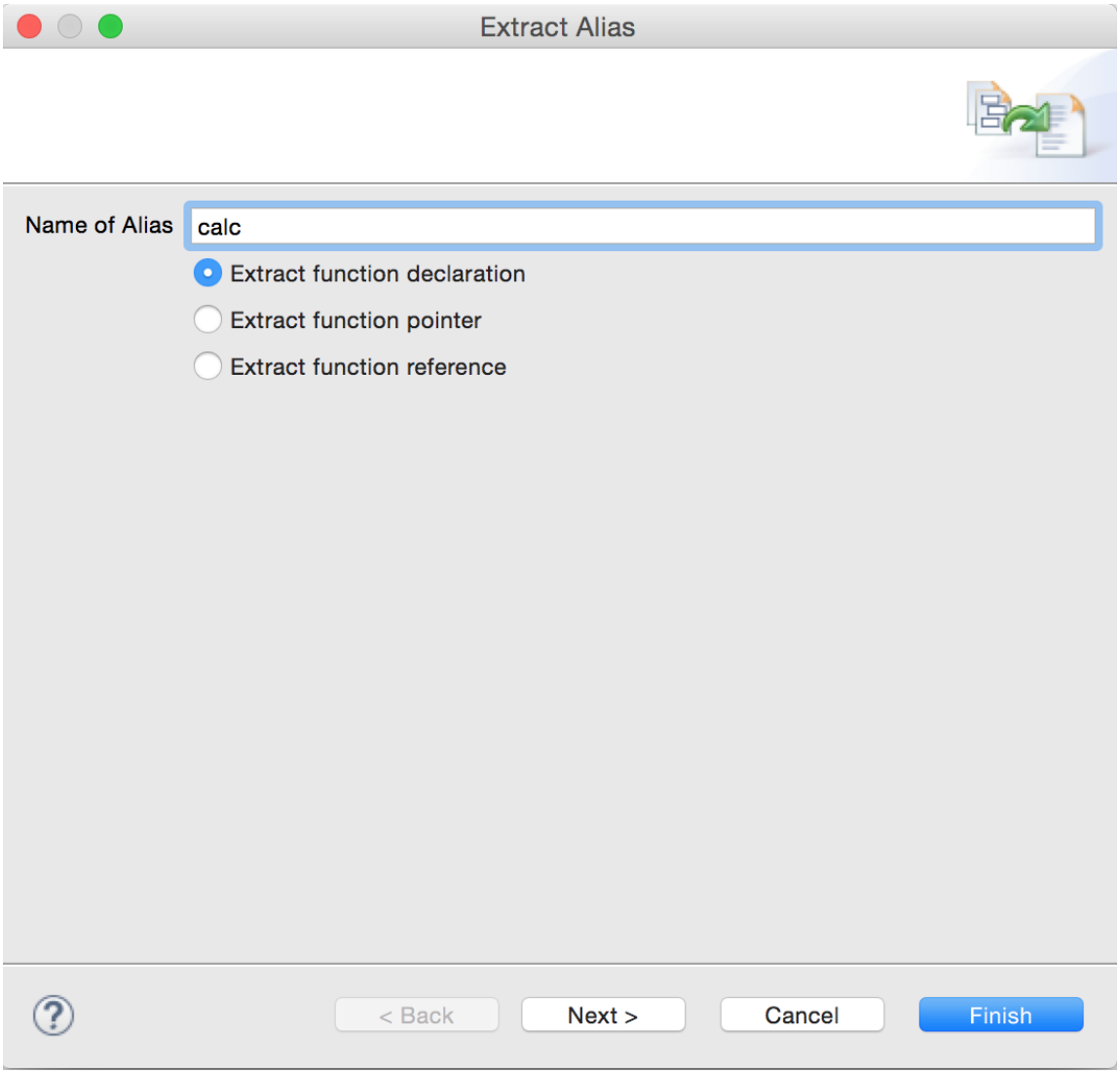

Figure 11: wizard for function declarations

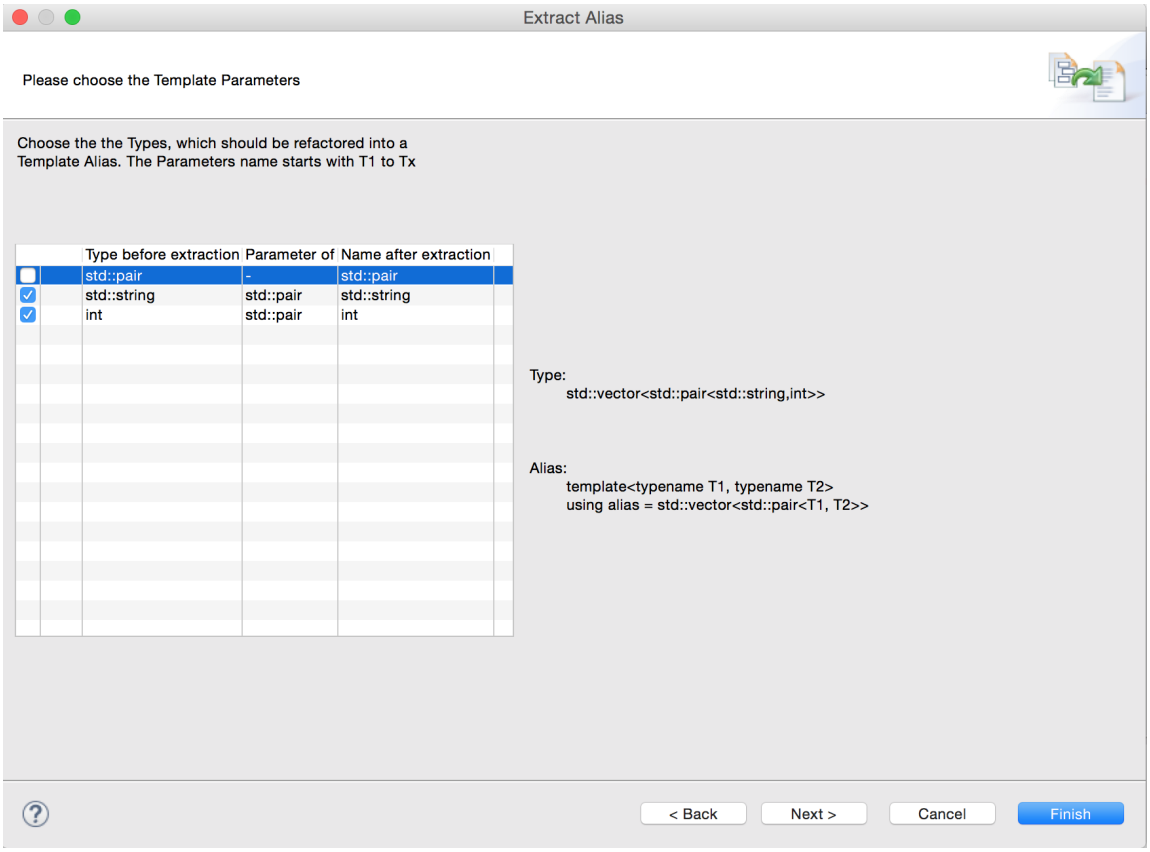

Figure 12: wizard for template parameters

And lastly of course we have the diff viewer from Eclipse:

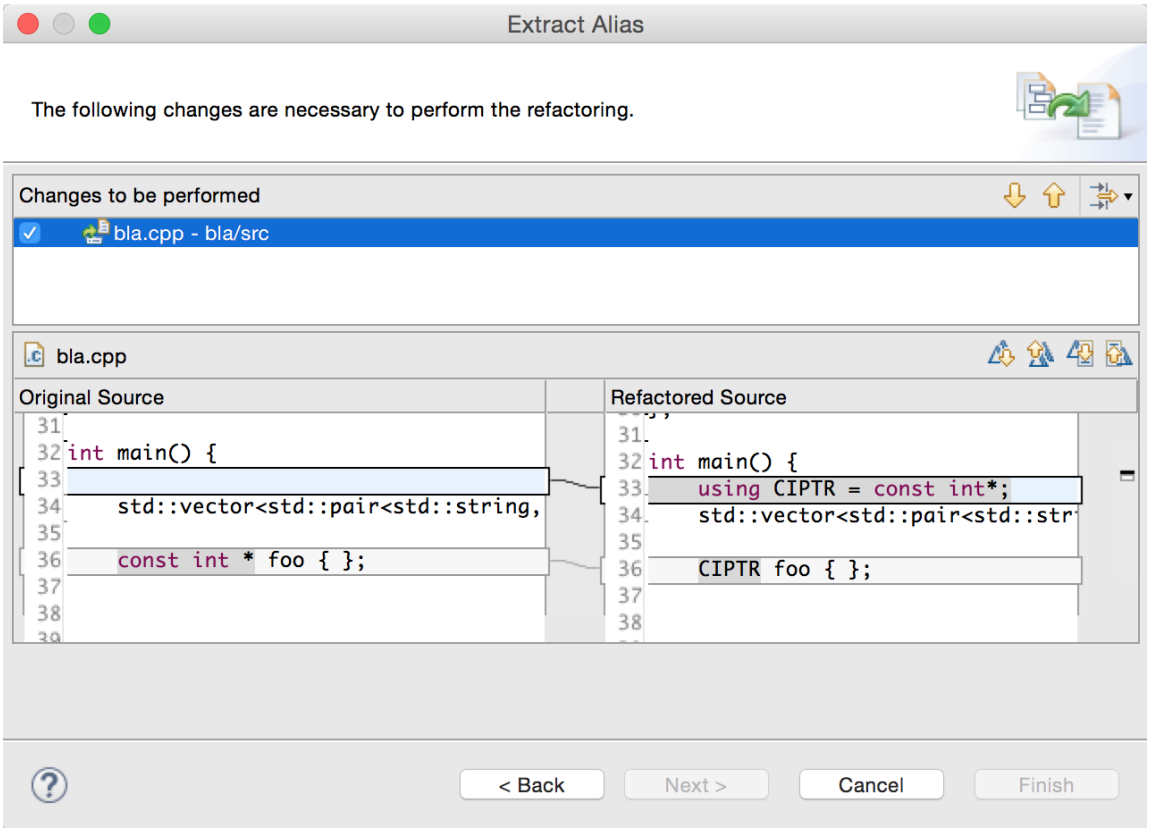

Figure 13: Eclipse diff viewer

6.3 Refactorings

When developing an Eclipse CDT plugin, every refactoring you implement extends from CRefactoring. But as discussed earlier in 6.2.1.4 on page 27 we built a little architecture, with a mix of the proxy as well as the strategy pattern. In this part we will have a look at the more specific strategies and their internal logic.

6.3.1 SimpleRefactoringConcreteStrategy

This refactoring strategy focuses on normal (complete) node selection. This was the first refactoring we implemented since it mainly focuses on shifting around nodes

and checking equality of each nodes. Also we will introduce which three refactoring cases we want to handle in our SimpleRefactoringConcreteStrategy and explain why it makes sense to use a shared logic for these three cases.

1. Selection of a DeclSpecifier
2. Selection of a DeclSpecifier with all its pointer operators
3. Selection of a TypeId (DeclSpecifier with an abstract Declarator)

```
int main() {
  using CI = const int;
  CI foo { };
}
```

Listing 28: DeclSpecifier (1)

```
int main() {
  using CIPTR = const int * const;
  CIPTR foo { };
}
```

Listing 29: DeclSpecifier with pointer operators (2)

```
#include <vector>
int main() {
  using letter = std::string;
  std::vector<letter> strings { };
}
```

Listing 30: TypeId (3)

Here's also the AST from listing 28 and 29. We haven't included a AST representation from listing 30 since it only makes it more confusing than actually helping to understand it.

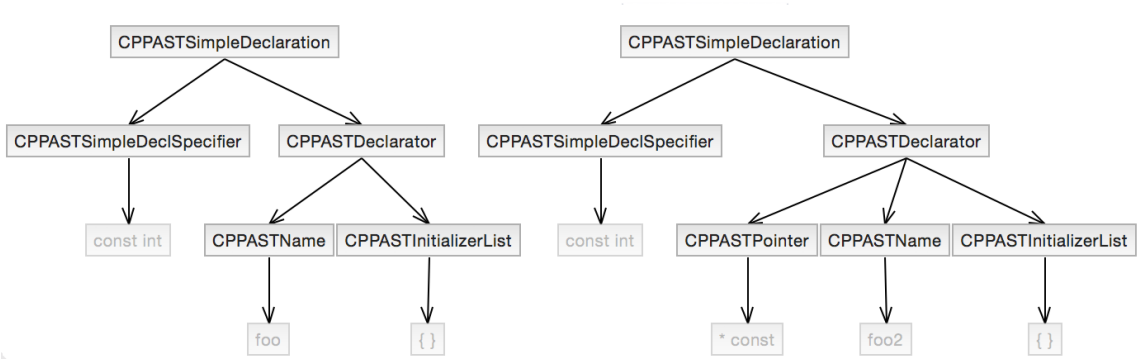

Figure 14: AST of simple refactoring cases (1) & (2)

6.3.1.1 Selection

```
const int * const foo { };
```

Figure 15: Selection of listing 28

In the first example (listing 28) we only selected "const int", which is a CPPASTSimpleDeclSpecifier. Detecting the selection of this case is pretty easy since the entire selection area is exactly the same of what the node includes. For cases like these, we can use the code example of listing 31.

```
int selectOffset = selectedRegion.getOffset();
int selectLength = selectedRegion.getLength();
IASTNode enclosingNode = ast.getNodeSelector(null).findEnclosingNode(selectOffset,
    selectLength);
```

Listing 31: findEnclosingNode()

```
const int * const foo { };
```

Figure 16: Selection listing 29

When we take a look at the second example though, the selected part is "const int * const" which is, when we use our findEnclosingNode() function, a CPPASTSimpleDeclaration. This is perfectly fine since we still know what to do with it once we come to our refactoring logic.
In the third example we want to select "std::string", which can be either a CPPASTTypeId, CPPASTNamedTypeSpecifier or a CPPASTQualifiedName, since the content of all the nodes are the same. What we actually have to do in our selection, in this case, is to loop upwards in our AST until we reach a CPPASTTypeId since this makes our final refactoring code easier.

6.3.1.2 Refactor logic

In this section we will only cover what the refactoring task consists of and how it's implemented. Things like the AST visitor section 6.5 on page 44 or wizard in section 6.2.1.6 on page 30 will be discussed in a separate section.
Creating a type alias is pretty straight forward for our three refactoring cases, which we explained in section 6.3.1.1 on page 36. We simply have to make a copy of the selected node, create a new type alias, find the type id of our selected node and add it to our created CPPASTAliasDeclaration. Then we have to find out which enclosing scope we are in (see 6.4 on page 44) and the insert it. That's it!

```
@Override
public void collectModifications(ASTRewrite rewrite) {
  char[] aliasName = proxy.getTheUserInput().toCharArray();

  change(rewrite, aliasName);

  IASTNode aliasStatement = createAliasStatement(aliasName);

  placeAliasDeclaration(rewrite, refactorSelection.getSelectedNode(), aliasStatement
    );
}
```

Listing 32: collectModifications()

What happens in our change(..., ...) method is kind of harder to explain. But we'll try to keep it as simple as possible. Depending on if the user selected to refactor multiple occurrences of the selected node, we visit our AST (only scope from selection) for several occurrences and add them to a list (see BaseASTVisitor). Since this refactoring involves three cases, as discussed in section 6.3.1.1 on page 36, our logic also includes some type checking to make any kind of refactoring possible. To implement this functionality we had to find a reliable way of checking for equality

of nodes where the Eclipse CDT wasn't able to provide a simple help. We basically had to implement a few equality checks in our ASTHelper, which depending on what type of node we wanted to compare, was not very simple.
Another thing we had to do here, is to remove storage classes of a declaration since they are not allowed in a type alias. Apart from that in this refactoring logic, we had the luxury of being able to work with nodes and easily replacing them unlike with partial selection of nodes which we will discuss now.

6.3.2 PartialDeclSpecRefactoringConctreteStrategy

In this section we will talk about partial selection. The idea of partial selection is to cover cases like in listing 33.

```
int main() {
  using score = int;
  const score ai { };
}
```

Listing 33: simple partial selection example

Before doing the refactoring of a partial selection the AST looked like figure 18.

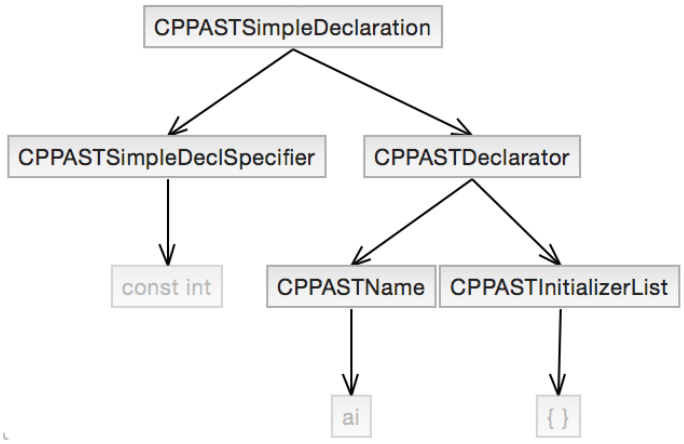

Figure 17: AST of const int

It's clear to see that we can't use our findEnclosingNode() method, which we discussed earlier anymore since what we want to select is "int". And this is where a

lot of problems started to occur. Also the NodeSelector from the CDT which we used earlier wasn't able to provide us any help and since we were only able to get the selected part of a node by string, we were doomed to implement an own way to handle selection. Partial selection of a node makes sense in both DeclSpecifier as well as Declarator. For example, if we have a "const int * const *" it makes sense that the user only wants to select "int * const".

6.3.2.1 Selection

Our goal was to implement a partial selection of nodes, which will work in all cases without relying too much on inelegant string comparisons. Listing 34 shows an example, on which we wanted to support any type of partial selection so you can get a better understanding of what our task included.

```
extern std::vector<std::pair<std::string* const * volatile, int* *>> const bar;
```

Listing 34: more complex partial selection example

The question, if this exact example makes sense is something else. With the code of listing 34, we want to illustrate what type of partial selections are possible and have to be handled by our implementation. Not only had our code to be able to handle the partial selection correctly, but also we had to be able to find several occurrences of said partial selection in other nodes.
For now we only want to determine what was selected, if the selected part makes sense to be refactored into a type alias and ultimately figure out how to do the refactoring by creating the corresponding nodes. For this task we had several options which we considered.

Idea 1: remove attributes, add attributes, avoid data storage
Our first idea was to simply just remove the selected attributes from a declaration and add them to the alias declaration and replace each occurrence with the alias.
With this approach we for example have a "const int * const" where "int" is selected. "const int" itself is a SimpleDeclSpecifier, which has the property isConst() set. While creating a DeclSpecifier with an alias we actually have to create a NamedTypeSpecifier and apparently there is no way to make a NamedTypeSpecifier out of a DeclSpecifier. So no matter if we choose this approach we'd still have to create new nodes, which means we'd have to store data, like if a DeclSpec has to be const, has

a storage class or what type of specifier is set. Storing data like those makes even more sense since we also need those information when it comes to finding several occurrences of a selected node. So we decided that this easy way of approaching our problem won't work in the long run.

Idea 2: Use strings
The selection, which we have from our editor, can only be retrieved as a standard Java string. This led us to another idea of exclusively working with strings, which sounds scary at first but made a lot of sense, after we thought about all the equality checks we will need in partial selection of a node. But quickly we realized that this approach leads to more and more trouble since code formatting and placement of type qualifiers start to matter here. Also it didn't make much sense for us to go back to strings, which we tried to avoid as much as possible during the entire development process.

Idea 3: Use offsets. Implement an isInSelection(IASTNode node)
An approach Prof. Sommerlad advised us to consider was to work with the offset of our node and the offset of our selection. Using these two known variables we could implement an isInSelection() method, which we could use to determine if a node was inside our selection range. But this approach doesn't solve our problem at all, since we want to know if the part of a node is in the selection range which can only be parsed as either an IToken or a Java string. The idea to use the offset for helping purposes is good but doesn't fully serve the purpose. We'd still need to store data like type qualifiers and storage class and also we'd encounter more problems when it comes to finding several occurrences.

Our chosen approach
This part may be a bit more complex since the entire code is separated in three parts.

- Selection logic
- AST Visitor equality check for multiple occurrences
- Refactoring logic checks

The approach we have chosen to take is kind of a mix between idea 1 & 2. We decided to implement a separate class called PartialRefactorSelection, which has the

task to determine with the help of the selection string as well as the enclosing nodes, how the final refactoring should look like and store the prebuilt node parameters. It makes even more sense having this common logic stored in one central place, since we find several occurrences we want to determine how the final refactoring of those other occurrence (different for every occurrence) should look like as well and since all the nodes have to be rebuilt at a later stage as a NamedTypeSpecifier it also makes the refactoring code more simple when we already know what we have to create.
For the actual detecting of what part has been selected and storing them we used ITokens. Sadly, we were not able to fully move away from string equality since the getType() method in IToken doesn't give any valid results. The entire logic is stored in our prepareReplacement() and preparePointerOperators() methods. What actually happens in here is that we iterate through every enclosing node, determine if each token is actually selected and remove/add type qualifiers accordingly. Of course this part of the code has to run for every occurrence found via our ASTVisitor.
In our visitor we again have to check if the selected DeclSpec is the same (which apparently can only be implemented by removing all type qualifiers and storage classes and comparing the string representation of each) as well as comparing if the selected pointer operators match with each their type qualifiers.
Of course we also have to check if the type itself (for example "int", "Car", ...) is selected and add an error if not since a refactoring without said selection makes no sense. The only down side of this approach is, that in certain areas we still have to do some string equality and also we have to run the logic to determine what belongs to the new NamedTypeSpec for every occurrence, which can be quite some instructions when there's 10+ occurrences of a partial node. But after all with this approach we are able to compute all the variety of possible node constructions with any type of partial selection that actually makes sense and are also able to find multiple occurrences of the selection without the formatting of the given code playing any role in it (see PartialSelectionRefactoringTest (50+ test cases)).

6.3.3 FunctionRefactoringConcreteStrategy

In this section we will talk about alias extraction of a function to an alias. This is a separate concern since there are various type of extraction an user can desire. The user could want to extract the function declaration or the function pointer or even the function reference.

```
// initial situation
const int add(const int x, const int y){
  return x + y;
```

```
}

const int calc(const int x, const int y, const int (f)(const int, const int)){
  return f(x,y);
}

// possible extractions
using fdecl = const int (const int, const int);
using fptr = const int (*)(const int, const int);
using fref = const int (&)(const int, const int);

const int add(const int x, const int y){
  return x + y;
}

const int calc(const int x, const int y, fdecl f){
  return f(x,y);
}
```

Listing 35: Example of Function extracting

We leave the decision to the user. Once he decided what function extraction he wants we let our refactoring do the work and create the right alias statement and replace the occurrence with the alias in the right way which isn't too hard of a task. It is simply the logic of the 6.3.1 on page 34, since we don't have any special nodes.

6.3.4 TemplateAliasRefactoringConcreteStrategy

This part was very tricky to implement. The biggest problem was to scan the selected type and get all the template parameters, which also could have template parameters themselves and also could occur several times in the whole type. During the implementation a problem was solved and three more occurred, which wasn't very fun. And fixing them led a long debugging. But finally the implementation could be done.

```
// initial situation
int main(){
  std::vector<std::pair<const std::string, std::pair<const int, std::string>>>
    keyKeyValue { };
}

// possible extractions
template<typename T>
using alias = std::vector<T>;

template<typename T1, typename T2>
using alias = std::vector<std::pair<T1, std::pair<int , T2>;

template<typename T1, typename T2>
using alias = std::vector<std::pair<T1, T2> , std::string>;
```

```
// and so on...
```

Listing 36: Example of alias template

Now, first of all we stored all the Names from the type. Once we have access to the names we have also access to the type as node itself. To store all the names of course we used the visitor, which visited all IASTNames. As a first thought we stored the names in the list and if we want to put them in the alias statement we just could simple take two elements from the list and thats it, we have the fully qualified name. But what it's not a name but a SimpleDeclSpecifier like int? Or a fully qualified name in a namespace like in the listing 37.

```
namespace foo {
namespace bar {
  template<typename T>
  using map = std::vector<std::pair<int, T>>;
}
}

int main()
{
  foo::bar::map<std::string> myMap {};
}
```

Listing 37: Example of alias template with namespace

So to make sure a type ends we separate them in the list by adding a null reference. We can now iterate over the list and whenever a null reference appears the type ends. This problem is solved.

Another problem is in the listing 36 on page 42 the type std::pair appears two times and so does string. That means we also have to store more information about the type then only its name. We have to make sure now which type as template parameter is in the whole type. To do that we created a datatype Pair, which stores additionally to the type the occurrence of the type. The first occurrence will store the type and zero, as an integer, as occurrence and the integer will be incremented by the occurrence of the type. This problem is also solved.

The user now is able to select which types he would like to replace in the alias template by a parameter T. And this leads to another problem. And the problem is that a template parameter can also have template parameter itself. Like the pair in the vector from the listing 36 on page 42. So if the user wants to replace the pair with a template parameter it doesn't makes any sense if its template parameter would stay there. The user could input a type with no template parameter at all and this will

lead to compile errors. So we have to remove them also. But we don't actually know which types are included as template parameter. We could store this information too but since the alias statement is build recursively which actually means we iterate over the type an replace the selected names with template parameter this can be done during the creation of the statement. This problem is also solved.

The last problem was a small problem. We also had to store the selected nodes to place them as template parameter in the new created alias. Since we began in the last 4 weeks with this feature, and it took lots of time we were not sure if we could implement it fully working, but after some big struggles we finally made it to work properly.

6.4 Scoping

It makes sense to have a type alias in the same scope as the selected node so we didn't find it necessary to implement a way for the user to decide in which scope he wants to have the type alias to be created. We implemented a simple method in our ASTHelper class, which returns the surrounding scope of a given node (also works with global scope). This can not only be useful to determine where our new type alias has to be placed, but also if we want to find several occurrences of a node inside the same scope using a AST visitor.

```
public static IASTNode findEnclosingCompoundStatementOrTranslationUnit(IASTNode node
    ) {
  while (!(node instanceof ICPPASTCompoundStatement || node instanceof
    IASTTranslationUnit)) {
    node = node.getParent();
  }
  return node;
}
```

Listing 38: findEnclosingCompoundStatementOrTranslationUnit()

6.5 AST Visitor

We have implemented a base visitor class called BaseASTVisitor. The purpose of this visitor is, that we have a storage of same node occurrences, which can easily be retrieved via getter. This is a necessity which every visitor has since we support finding several occurrences for every type of selection. The visitors themselves are

pretty much self explanatory and we've already covered pretty much everything about them in the package description 6.2.1.2.

6.6 ASTHelper

The ASTHelper is a creation of us containing some logic which is needed either in the refactoring or in the visitor.

6.6.1 Helper methods for the visitor

The visitor itself does not need any fancy logic. It just visits nodes, and has to decide, if this node is equals to the selected node (the node, which we want to extract in an alias). Sounds simple, but the part with the decision if an node is equal was very tricky since the Eclipse CDT had no simple equality for nodes implemented. An own implementation was required. In listing 39 is a quick example on how the content of the ASTHelper looks like.

```
public static <T extends IASTNode> boolean isSameIASTDeclSpecifier(T node, T
    selectedNode) {
  if (ASTHelper.areSameType(node, selectedNode, Type.ICPPASTSimpleDeclSpecifier)) {
     return ASTHelper.hasSameSimpleDeclSpecSignature((ICPPASTSimpleDeclSpecifier)
    node, (ICPPASTSimpleDeclSpecifier)selectedNode);
  } else if (ASTHelper.areSameType(node, selectedNode, Type.
    ICPPASTNamedTypeSpecifier)) {
     return hasSameNamedTypedSpecSignature((ICPPASTNamedTypeSpecifier)node, (
    ICPPASTNamedTypeSpecifier)selectedNode);
  }
  return false;
}
```

Listing 39: The isSameIASTDeclSpecifier

```
public static <T extends IASTNode> boolean hasSameCPPASTDeclaration(T node, T
    selectedNode) {
  if (areSameType(node, selectedNode, Type.CPPASTSimpleDeclaration)) {
    return isSameSimpleDeclarationSignature((IASTSimpleDeclaration) node, (
    IASTSimpleDeclaration) selectedNode);
  } else if (areSameType(node, selectedNode, Type.ICPPASTTypeId)) {
    return areSameTypeId((IASTTypeId) node, (IASTTypeId) selectedNode);
  } else { // Neither a SimpleDeclaration or TypeId
    return false;
  }
}
```

Listing 40: The hasSameCPPASTDeclaration

Both methods just check the type of node and calls the right method to check. To implement the equals a lot of analyzing was necessary. We had to analyze the structure of all these types in order to first of all find them quickly and to now what we are able to do with this types.

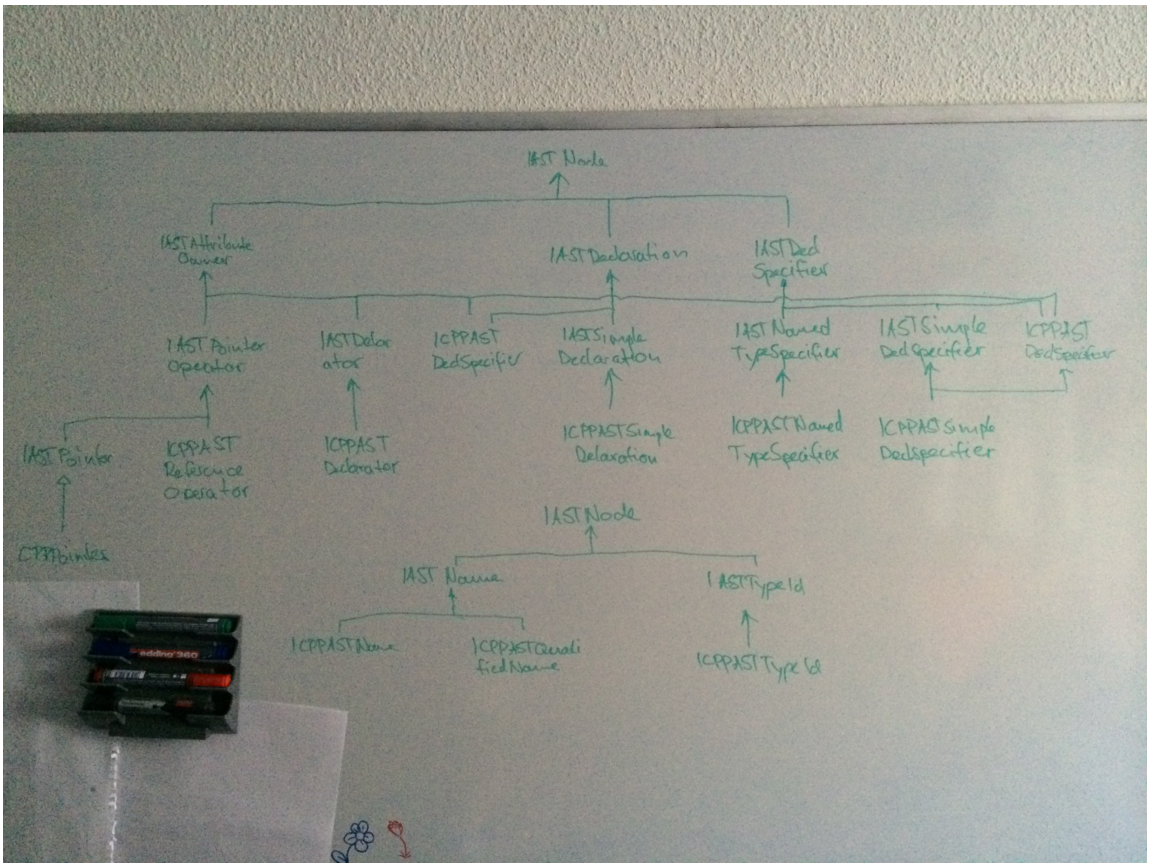

Figure 18: Constellation of nodes

After the analysis it was a bit clearer, what can be done with this types and where to find them and also it got clearer, why we only needed to visit very few type of nodes in the visitor.

6.6.2 ICPPASTSimpleDeclSpecifier

The equality check for SimpleDeclSpecifier was very easy. Because this type implements IASTAttributeOwner we used getType() to check if its the same type for example int or float and because it implemented IASTDeclSpecifier we were able to check if the type isConst() and isVolatile().

6.6.2.1 ICPPASTNamedTypeSpecifier

So this was an easy one lets get to the hard ones. The problem was that the NamedTypeSpecifier is also a subtype of IASTDeclSpecifier. And because of the big information a NamedTypeSpecifier could contain we were a bit concerned about how to do a right equality check. We could visit it with a special designed visitor to find all Names and SimpleDeclSpecifier and check them if they are equal. But this would involve a lot of coding. Then we found a solution (thanks to our troubleshooter Toni). We could use getName().resolveBinding(). And this had an equals implemented. The good thing was, it also knew if an alias was equals to it because of its name.

6.6.3 ICPPASTDeclaration

The Declaration is a bit trickier. As discussed earlier, a Declaration always contains a SimpleDeclSpecifier or NamedTypeSpecifier and a Declarator. The equality of SimpleDeclSpecifier and NamedTypeSpecifier is no problem, we already have a method which checks that, we just have to getDeclSpecifier() and call it. We just need to check the Declarator. For the Declarator we needed to know what it could contain and we found out that it could contain a ReferenceOperator or multiple PointerOperator and a Name, in which we are not interested because we don't touch it with our refactoring. So we had to check if it contains a ReferenceOperator with Pointers or does it contain only Pointers, because there are no pointers to References. A ReferenceOperator has no special Attributes like const or volatile but pointers do, so we just need to check if the both are References. And then we loop through the Pointers and check if they are same const and volatile.

7 Conclusion

7.1 Achievements

Our test cases pretty much describe the features of our plugin. But since they are not very intuitive we'll list them in this section:

- Normal selection of a node (DeclSpecifier, SimpleDeclaration or TypeId)
- Partial selection of a node with or without pointer operators
- Extraction of function declaration, function pointer or function reference
- Extraction of an alias template from a simple type, which has template parameters, where one can choose which types should be parameters in the alias template.

7.2 Future work

Although we tried to implement as much features as possible into our refactoring plugin, there are still some parts we had to leave out due to the lack of time. In this section we will explain, which features aren't fully implemented yet and how our plugin could be improved with new functionalities in general. Since our term project was open scoped we also have some additional features which can also be implemented (mainly inspired by Prof. Sommerlad).

- Extraction into header file
- Quick type alias generation (like rename refactoring)
- Strong type alias

If one wants to improve our plugin there are various entry points which are of relevance:

Selection and Wizard data
Both selection and data from the wizard are stored and handled in our **AliExtorRefactoring** class. Reason for that since these are both knowledge needed from all the refactoring strategies. If you happen to come up with a new selection type the

getASTNodeBySelection() method of our **AliExtorRefactoring** class is the place to implement it.

New refactoring strategy
Depending on given circumstances, a different refactoring strategy has to be called. This is handled in our **ProxyRefactoring** class. Visiting of the AST as well as the error detection (circumstances which would make a refactoring impossible) occur in our checkTheInitialConditions() method. And finally in the collectModifications() method you can make changes to the AST. In here, options from the wizard should be considered as well (like refactor several occurrences). For an easy understanding you might want to look into our **SimpleRefactoringConcreteStrategy** class.

New AST visitor
New AST visitors should extend from our **BaseASTVisitor** class. Reason for that being to store several occurrences of the search node.

New wizard page
If one wants to add a new wizard, you should also add it to the page flow located in our addUserInputPages() method inside the **Wizard** class. You can also extend from our **BaseWizardPage** class or just have a look at the existing wizard pages to have an easier understanding of how to implement a new one.

New helper methods
Currently all our helper methods (like checking the equality of certain nodes) are stored in our **ASTHelper** class. So if you have a necessity for something which hasn't been implemented yet, this is the place do implement it.

7.3 Personal reflections

7.3.1 Özhan Kaya

My main goal with the term project in general was to learn new things, instead of doing repetitive work, which can become demotivating quite quickly. With our term project creating an Eclipse refactoring plugin for the Cevelop IDE I can say, that my

expectations were met. Our first 4 weeks felt more like a knowhow gathering time than actually implementing, but we quickly (to our surprise) got a working prototype up and running and our first refactoring was working by week 5-6. At that point we were very happy and the saying "den chömers ja grad abgeh" motivated us for the next few weeks. But quickly we realized that this was only the easiest part. In our weekly meetings we kept realizing what features can be added and which obstacles they brought with them. For example did it seem very easy to have a requirement of "I only want to select a type without a type qualifier" which turned out to be an interesting challenge. Usually, in a project with a final date I'm more relieved knowing how to implement something in advance but with this term project having open goals I didn't mind taking on a challenging task 70-80% in. Sadly we also invested quite some time "fighting" with the CDT to implement certain features (like determining a selection before we can initiate a refactoring), but it was nice to see that we're not the only group to have these struggles (team Constificator). But once our refactoring architecture was working and we could differentiate between partial and normal selection types it was nice to specify some additional test cases. In general our test cases were very helpful because it allowed us to quickly get an answer to questions like "can't we just implement X in this way?", "would it still work if we changed the selection detection to Y?" or "do we actually have to check if a node has children?". It happens that you loose track on some parts of your code with growing number of lines and you're not the only one working on it and this is where test cases were especially helpful. In general I don't regret choosing this term project and also our teamwork was flawless since I've already done quite a few projects with Kevin in the past.

7.3.2 Kevin Schmidiger

I was really hyped for this term project, because I always wanted to know how plugin development works. As I was taking the class for C++11 we of course, developed with the IDE Cevelop. But the formatter of CDT didn't allow to put a comment tag like in Java the @formatter:off/on tag. Then I was wondering how or what would be necessary to implement the feature for this kind of tags. And now with the implementation of our refactoring plugin we got an insight on how to implement a plugin. And a big plus in the project was that I love programming languages. Developing a refactoring plugin includes, that you need to get to know much about the language, which in our project was C++(11), which was very fun and also very interesting.

We had a bit of a starting lag, because it was not clear if I was allowed to the term project, but finally I made it and we could start our project. But after we started I was impressed how fast we got something to run so fast. The fact that we knew nothing about plugin development, but get something to run that quickly was cool. Then we had to gain more knowledge about the alias declaration. At first we thought, okay that's simple, but we quickly realized that there are way more possibilities to this than we first thought. But this was cool because we were getting more knowledge about the language and the project got more and more challenging discovering new possibilities. But it was not always a fun challenge. Setting the project up for the continues integration with the maven tycho plugin was a bit tricky, that was the reason for the users guide of setting the project up. Then the bigger challenge was the implementation of the alias template. At first I thought this would be at least easier as I thought, but still not easy. And after beginning of the implementation I was getting more and more problems and lots and lots of debugging was necessary to find the cause of the problem. But finally I was able to get it to work, which I am very proud of.
Also it was very good that we had meeting every week. So we could always ask questions, which we didn't need the answers instantly but we needed to clear for the further week. There prof. Sommerlad could give us input about which features we could add to the project and Toni was the code troubleshooter.
It was also very good, that we always could walk to the IFS and get help if we were struggling with problems, which needed to be cleared.
The quintessence of the project is, that I learned a lot about plugin development and C++(11) and since I originally don't have a computer science background I also learned a lot in general about developing a software. My partner Özhan was also a good choice, because we know each other very well and working as a team was very easy. We always knew what to do without bigger discussions and splitting the work was never a problem. It worked just fine.

A User Manuals

A.1 Demo Video

We've created a short video showing the main features of our plugin. This video is accessible on our final CD in the video folder as well as through a direct link on our server http://sinv-56012.edu.hsr.ch/

A.2 Project Setup for Eclipse Plugins

Implementing Eclipse plugins needs a well defined infrastructure. Setting the project up, was a bit tricky. We had to understand, how the tycho-maven-plugin works, which helps us for continuous integration, and how we have to configure it. We also had trouble getting it to work on the build server. At the end, where we finally got the build server to run with our plugin we thought we could make a documentation about it. So why not writing a user's guide for Project setup for Eclipse Plugins. So if you are on your way implement an Eclipse Plugin here's a step by step user's guide.

Note: This section is based on the Tutorial from Lars Vogella [Vog15b].

To get this tutorial to work you need two plugins. One is easily installable during the tutorial. Once you get a error message in the pom.xml file in the package tag, click on the marker and choose the option search for m2e connector. And the other one is from the IFS [IFS15b].

A.2.1 Project structure with the tycho-maven-plugin

Let's start at the very beginning. Implementing Eclipse plugins needs a given structure. In this section we are going to describe, how this structure should look like. We show how to build it step by step. We have to create some projects to get this to work. Each project has its own functionality. In each project we have to configure what its functionality is, which is explained later on.

A.2.1.1 Step 1: The parent project

At first we need an (at the beginning) empty project (Rightclick New → Project... → General → Project).

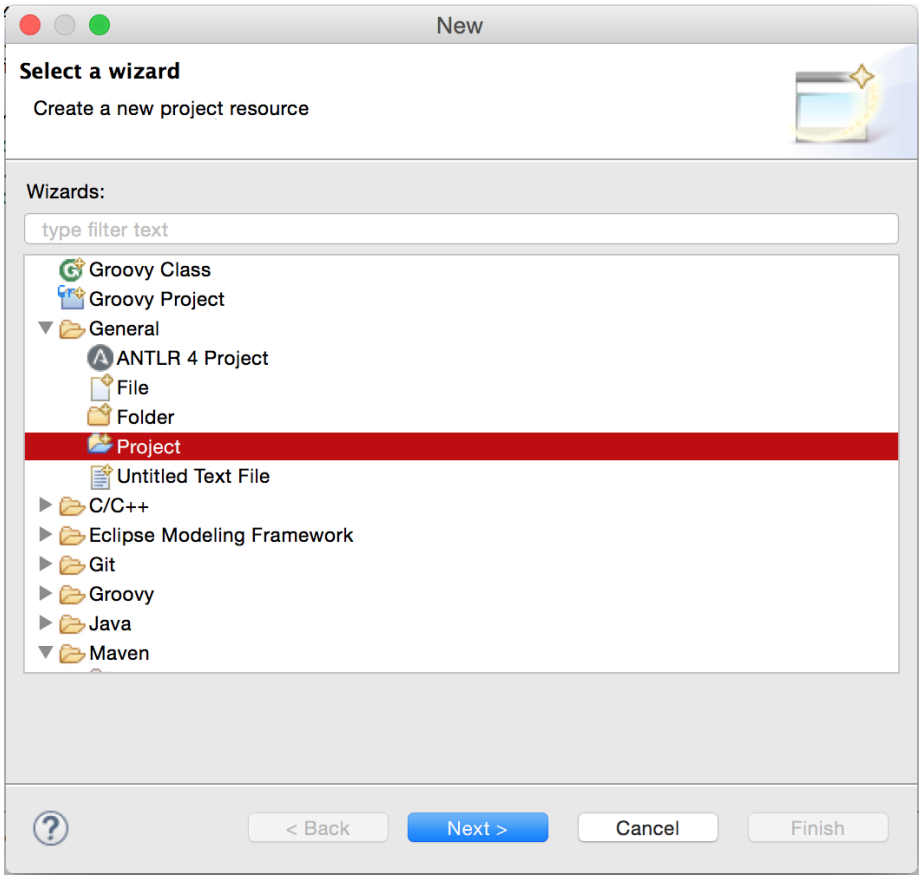

Figure 19: General project

Name it something like ch.hsr.ifs.testplugin.parent. In this project we have to create a pom.xml file. The pom file contains information about dependencies and other useful things, which is needed for continuous integration. You can do this manually or by rightclick on the project configure → convert to Maven project.

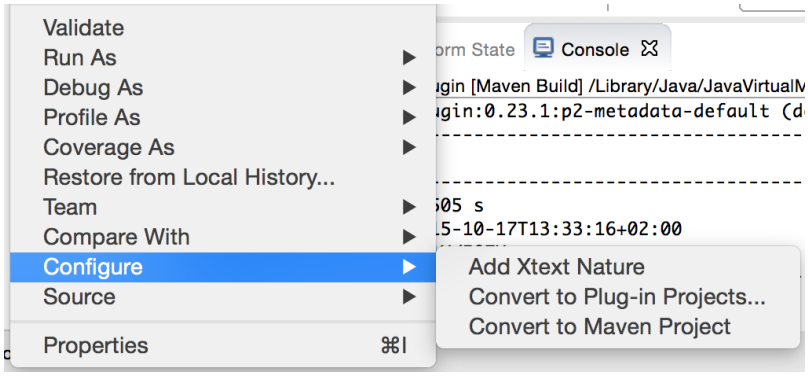

Figure 20: Convert to Maven project

Note:
It is also possible to create Maven projects instead of normal Eclipse projects. (Rightclick New → Other... → Maven → Maven Project).

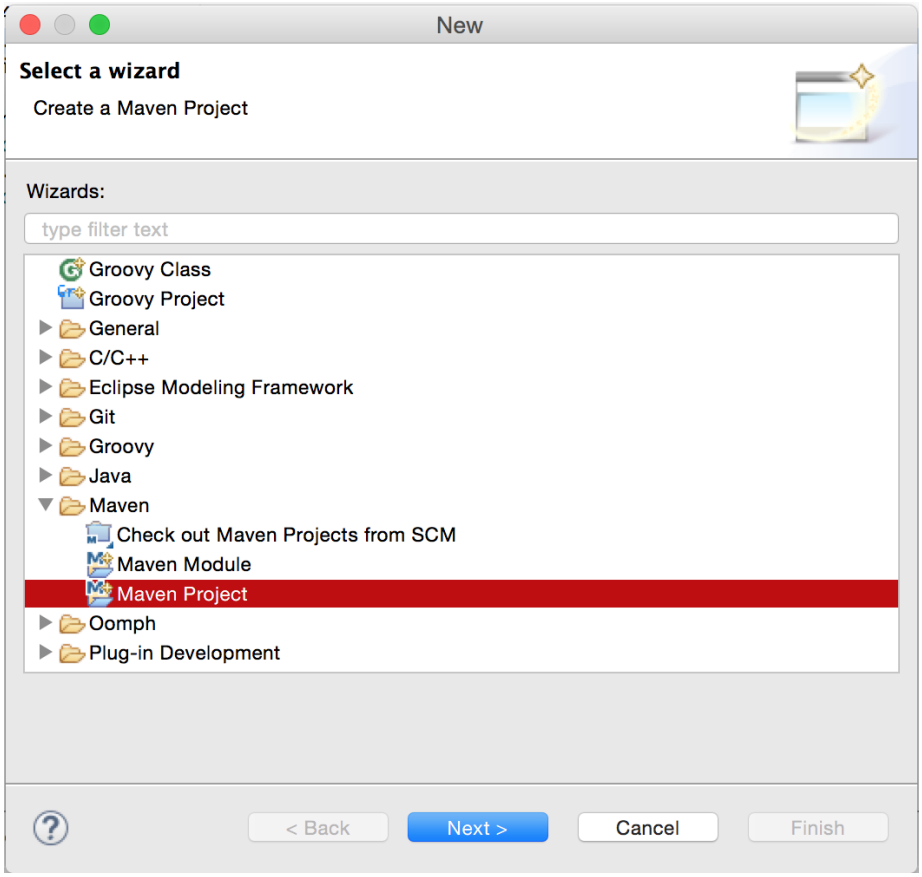

Figure 21: Maven project

The wizards creates a folder src, which is not needed, so just delete it.

Our pom.xml file contains the information from listing 41.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ch.hsr.ifs</groupId>
   <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
   <version>1.0.0-SNAPSHOT</version>
   <packaging>pom</packaging>

   <properties>
       <tycho.version>0.23.1</tycho.version>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
    <eclipse-p2-repo.url>http://download.eclipse.org/releases/mars</eclipse-p2-
 repo.url>
</properties>

 <repositories>
     <repository>
         <id>eclipse-p2-repo</id>
         <url>${eclipse-p2-repo.url}</url>
         <layout>p2</layout>
     </repository>
 </repositories>

 <profiles>
     <profile>
         <id>build-individual-bundles</id>
         <repositories>
             <repository>
                 <id>eclipse-p2-repo</id>
                 <url>${eclipse-p2-repo.url}</url>
                 <layout>p2</layout>
             </repository>
         </repositories>
     </profile>
 </profiles>

 <build>
     <plugins>
         <plugin>
             <groupId>org.eclipse.tycho</groupId>
             <artifactId>tycho-maven-plugin</artifactId>
             <version>${tycho.version}</version>
             <extensions>true</extensions>
         </plugin>

        <plugin>
            <groupId>org.eclipse.tycho</groupId>
            <artifactId>target-platform-configuration</artifactId>
            <version>${tycho.version}</version>
            <configuration>
                 <environments>
                     <environment>
                         <os>linux</os>
                         <ws>gtk</ws>
                         <arch>x86</arch>
                     </environment>
                     <environment>
                         <os>linux</os>
                         <ws>gtk</ws>
                         <arch>x86_64</arch>
                     </environment>
                     <environment>
                         <os>win32</os>
                         <ws>win32</ws>
                         <arch>x86</arch>
                     </environment>
                     <environment>
                         <os>win32</os>
                         <ws>win32</ws>
```

```
                            <arch>x86_64</arch>
                        </environment>
                        <environment>
                            <os>macosx</os>
                            <ws>cocoa</ws>
                            <arch>x86_64</arch>
                        </environment>
                    </environments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Listing 41: pom.xml in the parent project

A.2.1.2 Step 2: The Plugin Project

Now we need a plugin-project (Rightclick New → Other... → Plug-in Project).

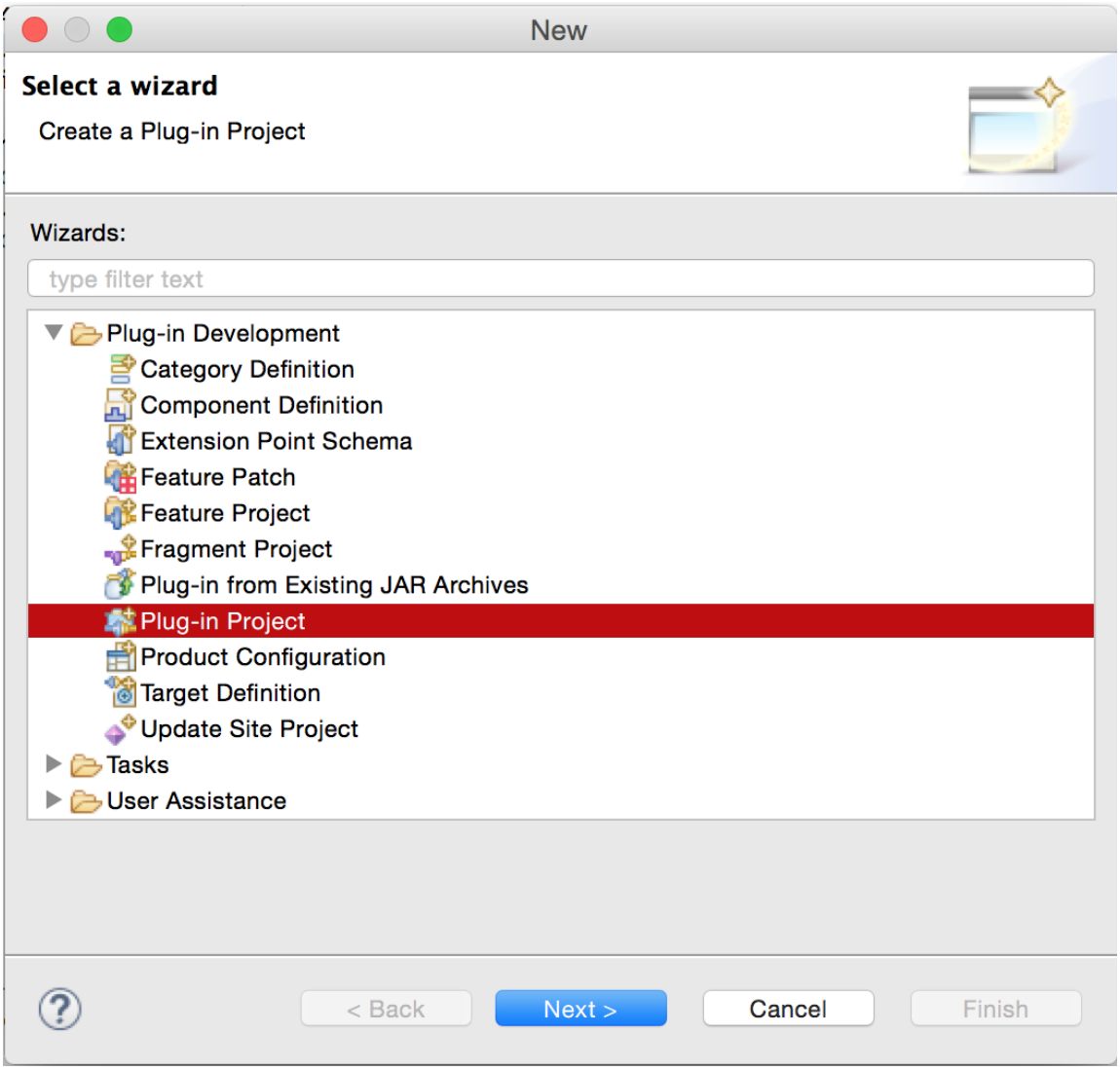

Figure 22: Create plugin project

Name it something like ch.hsr.ifs.testplugin. Remember, we don't need any code at first, we just build up the structure.

Note:
Here it is necessary to create a plugin-project and convert it manually to a Maven

project. (Rightclick on the project configure → convert to Maven project).

This project should look something like in figure 23.

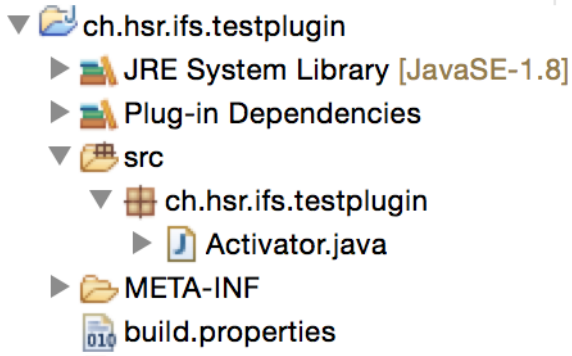

Figure 23: Plugin project

Then again create the pom.xml file, with the information in listing 42.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ch.hsr.ifs</groupId>
  <artifactId>ch.hsr.ifs.testplugin</artifactId>
  <packaging>eclipse-plugin</packaging>

  <parent>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>
</project>
```

Listing 42: pom.xml in the testplugin project

Note:
You may have to refresh the plugin-project. You can do that with rightclick on the project → maven → update project....

Note: If you converted the project you may have some errors in the pom.xml. Then you have to set the group id tag in the parent tag. Also you may have so

set the version tag under the artifactid. Then the errors should disappear and some warnings appear, where you can delete some duplicate code.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ch.hsr.ifs</groupId>
  <artifactId>ch.hsr.ifs.testplugin</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>eclipse-plugin</packaging>

  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>
</project>
```

Listing 43: pom.xml in the testplugin project

A.2.1.3 Step 3: Let's see if this works

Now we are going to check if everything works at this point. If you haven't done it yet, convert both projects so Maven projects, even if we already created the pom.xml. Now to check if everything worked right, rightclick on the plugin project → run as → maven build. If you do that the first time, you have to define the goals as follows:

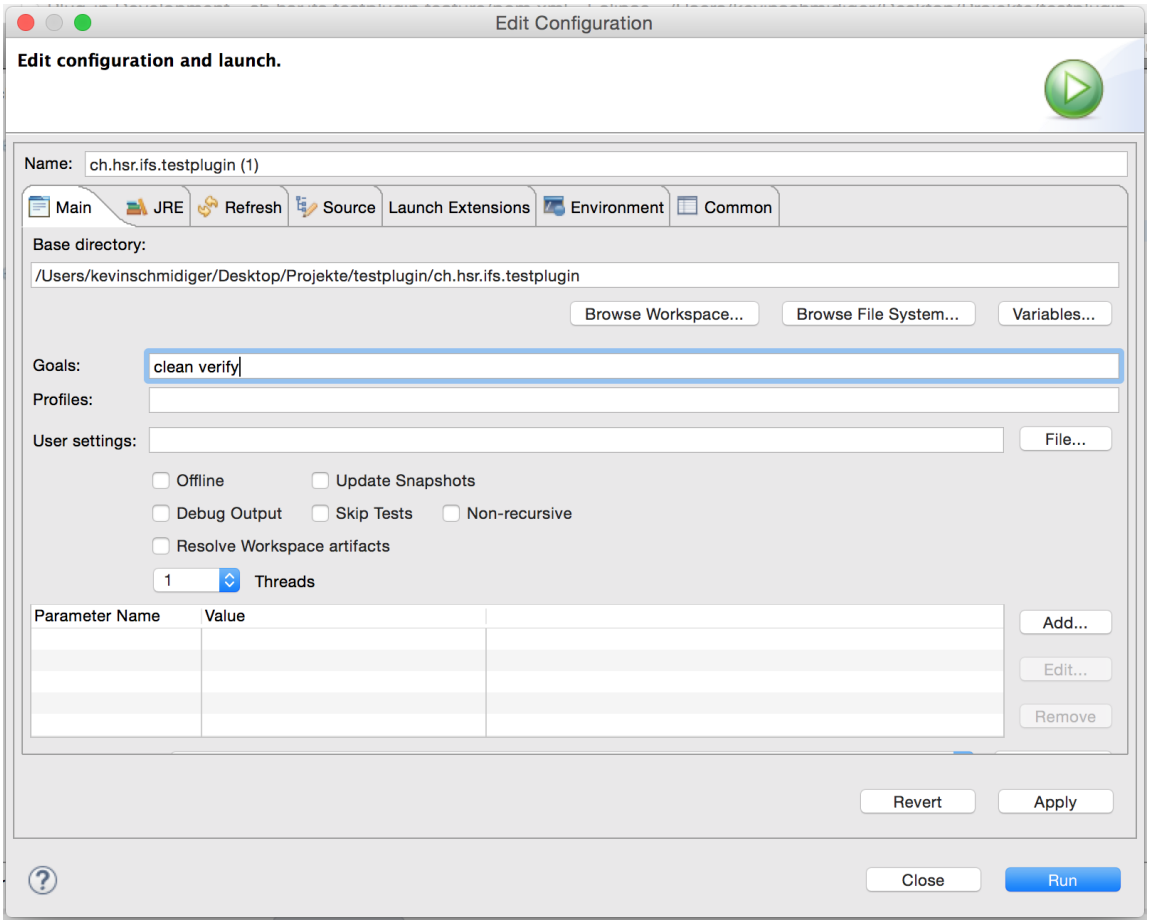

Figure 24: Maven goals

If everything worked fine the console should give an output similar to the following:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.505 s
[INFO] Finished at: 2015-10-17T13:33:16+02:00
[INFO] Final Memory: 71M/587M
[INFO] ------------------------------------------------------------------------
```

Figure 25: Maven build success

Note:
It should be clear that the time can vary.

A.2.1.4 Step 4: The Feature Project

In this step we have to create a plugin feature project. (Similar to Figure 22 on page VII). This is needed to show the features of your plugin to the users, which can be installed [Cod15]. Name it something like ch.hsr.ifs.testplugin.feature. The rest can be left with its default values.

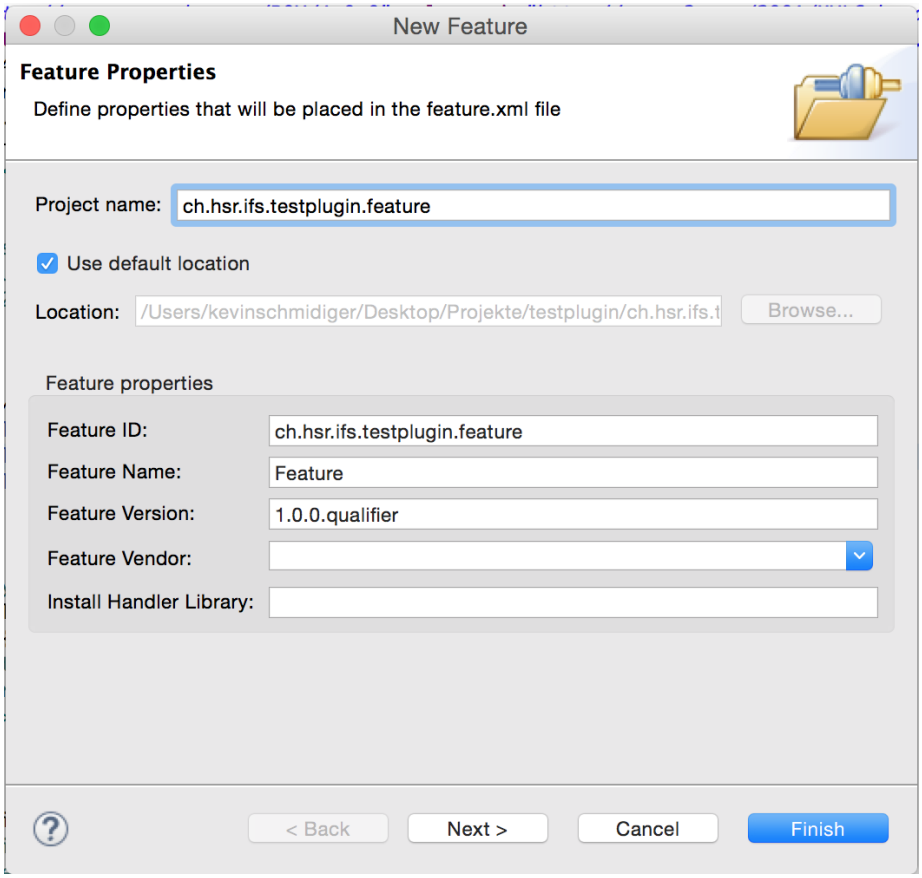

Figure 26: Plugin Feature Wizard

Click next. Now we have to reference our plugin. In this case ch.hsr.ifs.testplugin.

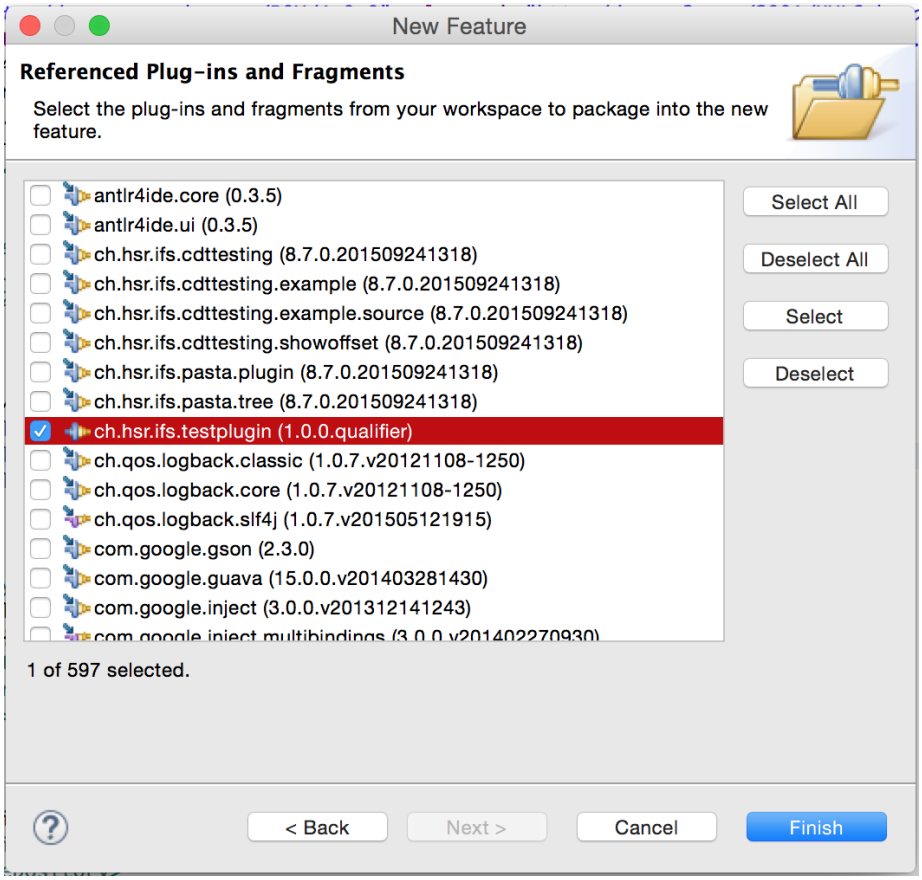

Figure 27: Plugin Feature Wizard page 2

Then click finish. That's it. Now we have of course to create the pom.xml file (see Figure 20 on page III).

The pom.xml file contains the information of listing 44.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>ch.hsr.ifs.testplugin.feature</artifactId>
  <packaging>eclipse-feature</packaging>

  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
```

```
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>
</project>
```

Listing 44: pom.xml in the plugin feature project

A.2.1.5 Step 5: Add modules in the parent project

If you tried to run the feature project you should get an error like this:

```
[ERROR] Cannot resolve project dependencies:
[ERROR]   Software being installed: ch.hsr.ifs.testplugin.feature.feature.group 1.0.0.qualifier
[ERROR]   Missing requirement: ch.hsr.ifs.testplugin.feature.feature.group 1.0.0.qualifier requires 'ch.hsr.ifs.testplugin 0.0.0' but it could not be found
[ERROR]
[ERROR] See http://wiki.eclipse.org/Tycho/Dependency_Resolution_Troubleshooting for help.
[ERROR] Cannot resolve dependencies of MavenProject: ch.hsr.ifs:ch.hsr.ifs.testplugin.feature:1.0.0-SNAPSHOT @ /Users/kevinschmidiger/Desktop/Projekte/testp
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MavenExecutionException
```

Figure 28: Feature build fail

The problem is maven can't find the testplugin. We have to tell it where it can be found. This is done in the parent pom.xml. we add a tag in with modules, where we define our projects.

```
<modules>
  <module>../ch.hsr.ifs.testplugin</module>
  <module>../ch.hsr.ifs.testplugin.feature</module>
</modules>
```

Listing 45: Modules tag in parent pom.xml

This tag can be placed after for example the property tag. Now we can check if this works so far. To check that we run the parent project (see section A.2.1.3 on page IX).

A.2.1.6 Step 6: The Targetdefinition Project

In this project we define against what we are deploying. As usual we create a new general project. Name it something like ch.hsr.ifs.testplugin.targetdefinition. And then we we can add a target definition file (Rightclick on the Project New → Other...

→ Plug-in Development → Target Definition). Somehow it is important to name the target file the same as the project.

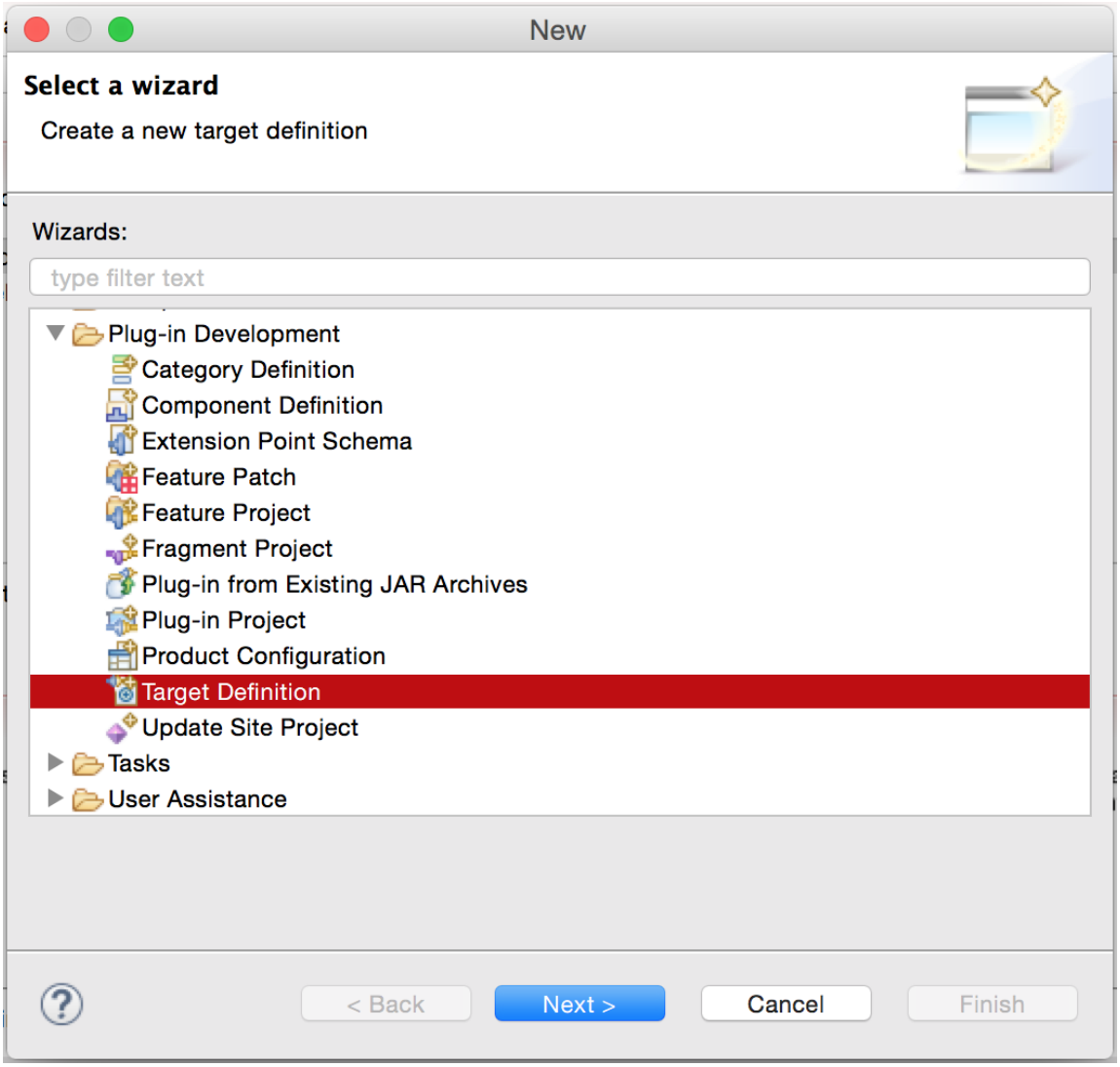

Figure 29: Wizard to choose target definition

We can now define against what we are deploying. In our case we are deploying against the Eclipse, the CDT and of course Cevelop. To add the targets we can do it manually or use the Eclipse editor. We are going to describe both ways, but it

is recommended to use the editor to make sure the versions are up to date. Open the target file in Eclipse with a double click. The editor will show you which targets we have but in our case it is empty. Click the add button and then choose Software Site. The wizard is similar to the wizard to install other plugins for the Eclipse IDE. Type or paste the Link: http://download.eclipse.org/eclipse/updates/4.5 [Ecl15c] into the textfield and wait until the wizard loads the repository. Then choose Eclipse Platform, Eclipse Platform SDK and Eclipse SDK.

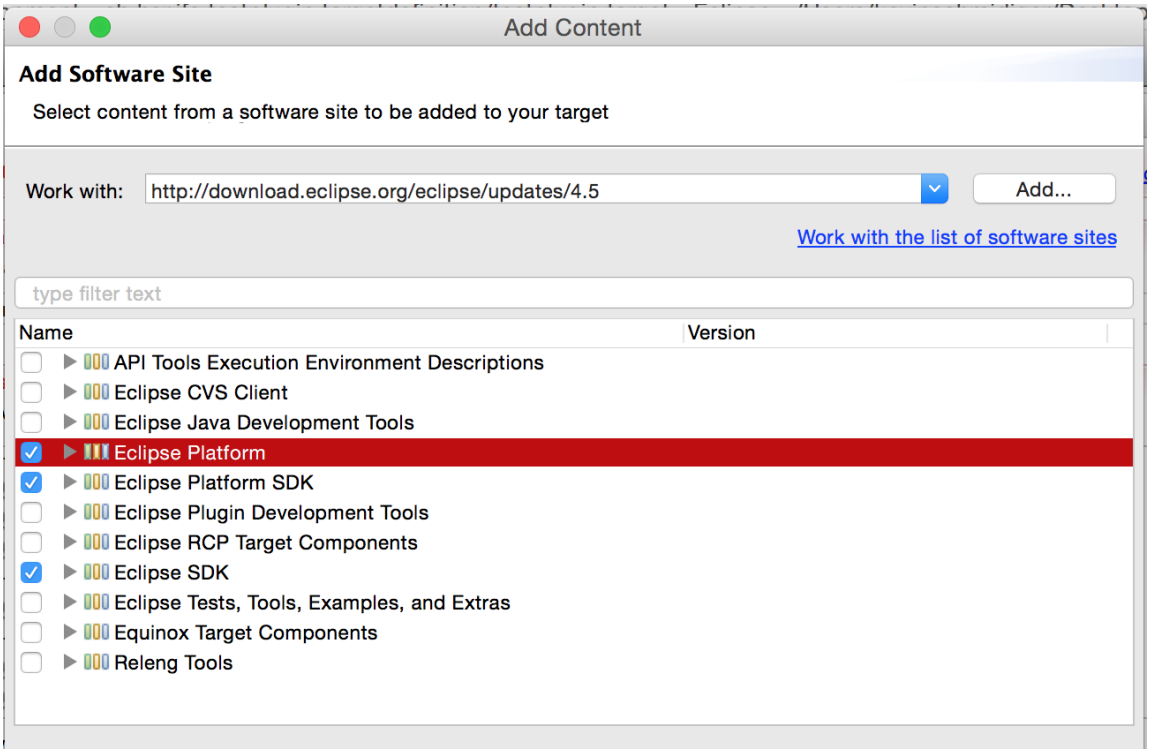

Figure 30: Select the eclipse SDK, eclipse platform and eclipse platform SDK

Repeat the steps with the Link: http://download.eclipse.org/releases/mars [Ecl15b] for the CDT. This time choose C/C++ Development Tools SDK under Programming Languages.

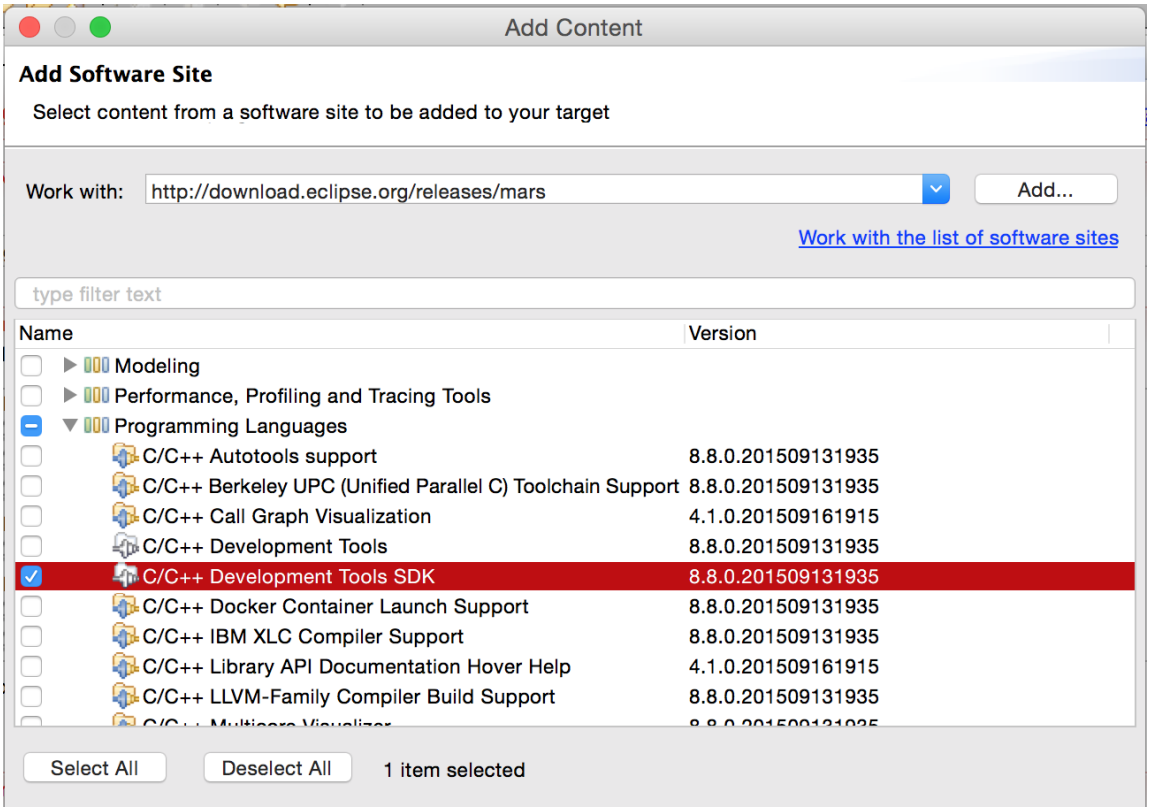

Figure 31: Select the C/C++ Development Tools SDK

And last but not least we add the testing features. The Link is: https://www.cevelop.com/cdt-testing/mars [IFS15b]. Choose the Testing Features and the we are done for the target file.

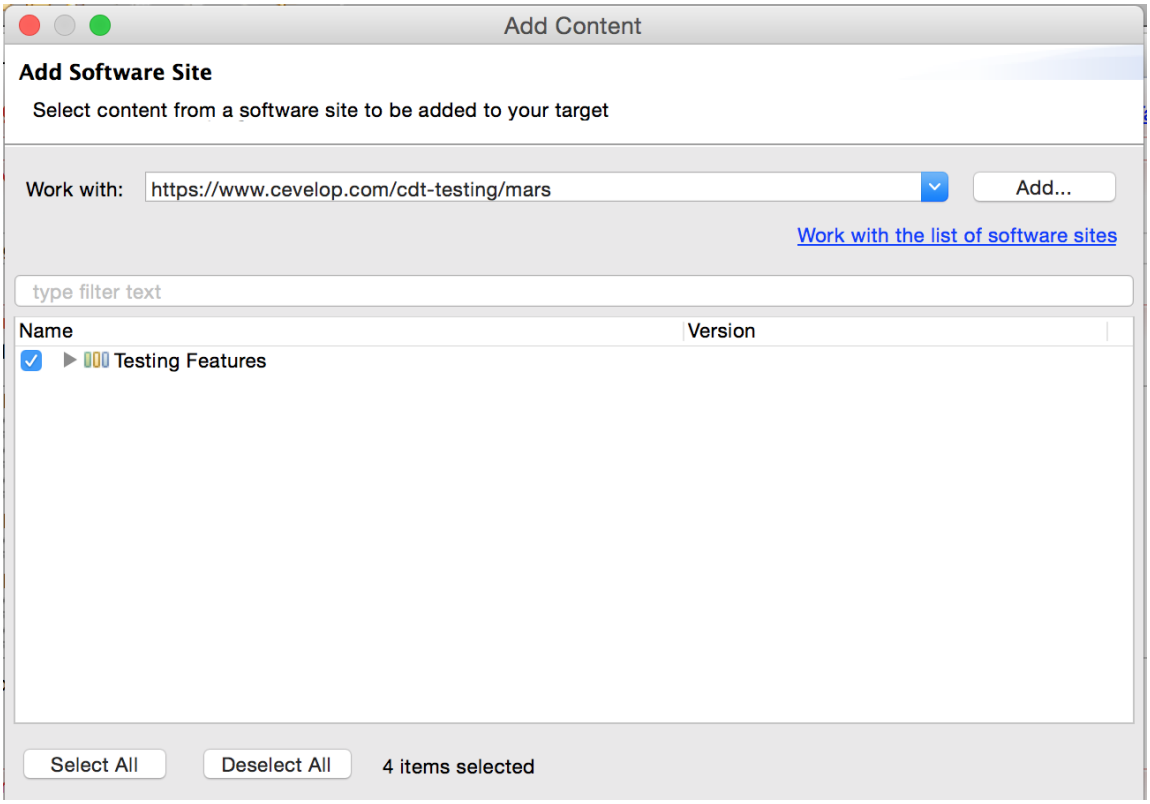

Figure 32: Select the Testing Features

Now if one wants to do that manually you have to open the target file in an editor (Eclipse has no editor for the raw file somehow so you have to choose one of your own). The file should look something like listing 46.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?pde version="3.8"?><target name="testplugin" sequenceNumber="-1"/>
```

Listing 46: testplugin.target

You have to change the target tag and add more locations to it.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?pde version="3.8"?><target name="ch.hsr.ifs.testplugin.targetdefinition"
    sequenceNumber="0">
<locations>
```

```
<!-- CDT -->
<location includeAllPlatforms="false" includeConfigurePhase="true" includeMode="
    planner" includeSource="true" type="InstallableUnit">
<unit id="org.eclipse.cdt.sdk.feature.group" version="8.8.0.201509131935"/>
<repository location="http://download.eclipse.org/releases/mars"/>
</location>

<!-- Eclipse -->
<location includeAllPlatforms="false" includeConfigurePhase="true" includeMode="
    planner" includeSource="true" type="InstallableUnit">
<unit id="org.eclipse.sdk.ide" version="4.5.1.M20150904-0015"/>
<unit id="org.eclipse.platform.sdk" version="4.5.1.M20150904-0015"/>
<unit id="org.eclipse.platform.ide" version="4.5.1.M20150904-0015"/>
<repository location="http://download.eclipse.org/eclipse/updates/4.5"/>
</location>

<!-- Cevelop -->
<location includeAllPlatforms="false" includeConfigurePhase="true" includeMode="
    planner" includeSource="true" type="InstallableUnit">
<unit id="ch.hsr.ifs.cdttesting.tools.feature.feature.group" version="
    8.8.0.201511261058"/>
<unit id="ch.hsr.ifs.cdttesting.example.feature.feature.group" version="
    8.8.0.201511261058"/>
<unit id="ch.hsr.ifs.pasta.feature.feature.group" version="8.8.0.201511261058"/>
<unit id="ch.hsr.ifs.cdttesting.feature.feature.group" version="8.8.0.201511261058"/
    >
<repository location="https://www.cevelop.com/cdt-testing/mars"/>
</location>

</locations>
</target>
```

Listing 47: testplugin.target with targetdefinitions

Note: As you can see we have to add the version numbers manually. This can lead to errors and if copy pasted you would maybe deploy against an old version.

Now, to get it to run with maven we need to add the pom file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>ch.hsr.ifs.testplugin.targetdefinition</artifactId>
  <packaging>eclipse-target-definition</packaging>
  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>
```

```
</project>
```

Listing 48: pom.xml in the target definition project

Don't forget to add the module in the parent pom.

```
<modules>
  <module>../ch.hsr.ifs.testplugin</module>
  <module>../ch.hsr.ifs.testplugin.feature</module>
  <module>../ch.hsr.ifs.testplugin.targetdefinition</module>
</modules>
```

Listing 49: parent pom.xml snippet

A.2.1.7 Step 7: The Test Project

Of course everybody wants to make sure the application works fine. With this project you can test your plugin. So once again we go and create a new project. But this time we need a normal empty plugin project. After the project is created we need to add some classes. We just explain the setup for this project for further explanation see [Rep15]. The structure looks something like figure 33.

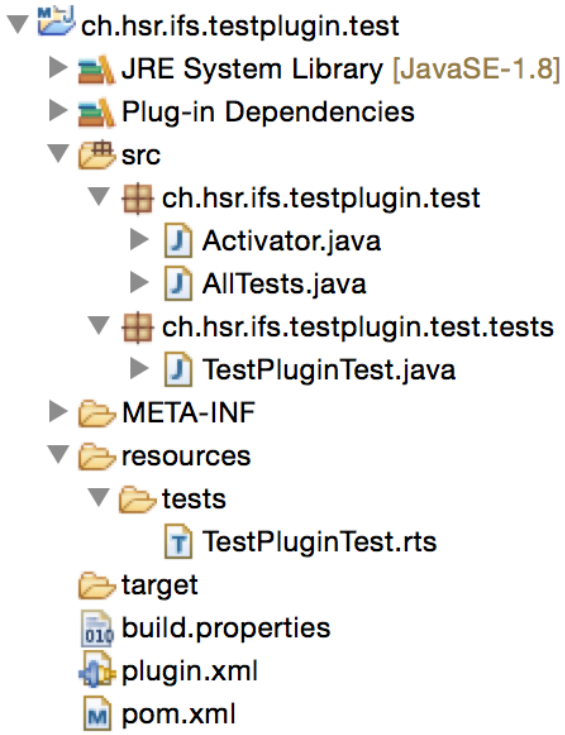

Figure 33: The structure of a test project

You will need the Activator to run the test project as a testplugin project and test your plugin. And the TestAll class to run you test classes.

And once again the pom file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>ch.hsr.ifs.testplugin.test</artifactId>
  <packaging>eclipse-test-plugin</packaging>
  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>

<properties>
        <tycho-version>0.23.1</tycho-version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
        <eclipse-p2-repo.url>http://download.eclipse.org/releases/mars</eclipse-p2-
    repo.url>
</properties>

 <profiles>
   <profile>
     <id>OSX</id>
     <activation>
       <os>
   <family>mac</family>
       </os>
     </activation>
     <properties>
       <swtbot.args>-Xmx1024m -XstartOnFirstThread</swtbot.args>
     </properties>
   </profile>
   <profile>
     <id>NotOSX</id>
     <activation>
       <os>
   <family>!mac</family>
       </os>
     </activation>
     <properties>
       <swtbot.args>-Xmx1024m</swtbot.args>
     </properties>
   </profile>
 </profiles>

 <build>
   <sourceDirectory>src</sourceDirectory>
   <plugins>
     <plugin>
       <groupId>org.eclipse.tycho</groupId>
       <artifactId>tycho-surefire-plugin</artifactId>
       <version>${tycho-version}</version>
       <configuration>
         <!-- Need UI harness for quick fix tests -->
         <useUIHarness>true</useUIHarness>
         <useUIThread>false</useUIThread>
         <testSuite>ch.hsr.ifs.cute.aliextor.test</testSuite>
         <testClass>ch.hsr.ifs.cute.aliextor.test.TestSuiteAll</testClass>

         <appArgLine>-nl en</appArgLine>
         <argLine>${swtbot.args}</argLine>
         <product>org.eclipse.sdk.ide</product>
         <application>org.eclipse.ui.ide.workbench</application>
         <dependencies>
           <dependency>
             <type>p2-installable-unit</type>
             <artifactId>org.eclipse.sdk.feature.group</artifactId>
           </dependency>
         </dependencies>
       </configuration>
     </plugin>
   </plugins>
 </build>
```

```
</project>
```

Listing 50: Test pom.xml

And don't forget to add it to the parent file.

If you want to try to check if everything works you have to skip the test project as long you didn't implement a refactoring and the test class (see [Rep15]). If both are implemented you can run Maven and it builds the projects and runs the tests. And if all went alright you should get a build success.

A.2.2 Getting it on Jenkins

We used Jenkins as continues integration build server. But to build the plugin you need to install the Xvfb [Xvf15] plugin [Jen15], which may also be needed on the sever itself, which we installed on our Ubuntu server [Hya15]. If those programs are installed you should be able to get it to run.

First we add a new job and select Maven project. Add the configuration of your source management. Choose the time of automated build.

Now the Maven and plugin dependent configurations. First the Xvfb.

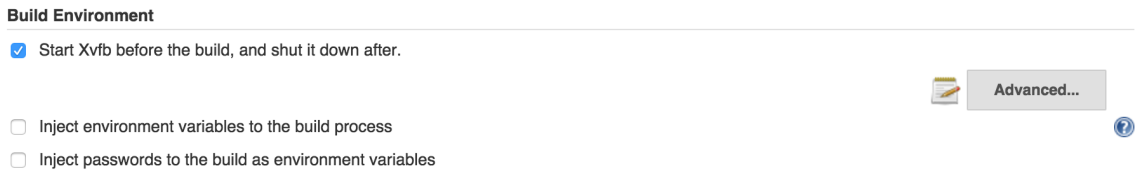

Figure 34: Xvfb configuration in jenkins

Then how Maven should build the plugin.

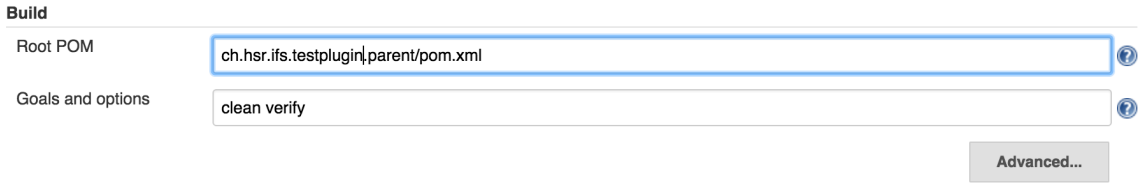

Figure 35: Parent pom.xml configuration in jenkins

You can add a mail notification as build setting step to stay informed if a build fails. But that's already it. Your server should build at a configured time.

A.2.3 Updatesite of the Plugin

The last thing is to make the plugin available for Eclipse or Cevelop. For this we have to add another project. This time its a plugin development where we select update site project. In the site.xml file we add a new category. It can be named Testplugin and the same for the ID. If you want you can add a description. To the category we add the feature of our plugin. And that's it.

Well, don't forget the pom file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>ch.hsr.ifs.testplugin.updatesite</artifactId>
  <packaging>eclipse-update-site</packaging>
  <parent>
    <groupId>ch.hsr.ifs</groupId>
    <artifactId>ch.hsr.ifs.testplugin.parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../ch.hsr.ifs.testplugin.parent</relativePath>
  </parent>
</project>
```

Listing 51: pom.xml of updatesite project

B Project organization

In this section we introduce our project management. We shortly explain which technologies were used und what our time schedule looks like.

B.1 Local Development Environment

The plug-in was created on two Macbook Pros with OSX Yosemite 64bit. And for the development we used the software listed in table 1.

Software	Version
Eclipse Mars	4.5
TEXShop	3.56
OpenJDK	1.8.0_20
Apache Maven	3.3.3

Table 1: Table of used Software and Version

B.2 Continuous Integration Server

The continuous integration system used in this project was provided as a virtual server sinv-56012.edu.hsr.ch hosted at the University of Applied Sciences Rapperswil (HSR). The operating system of the virtual server is Ubuntu 14.04.3 LTS 64-Bit. To build the AliExtor plug-in and related artifacts the software listed in Table 3 was installed.

Software	Version
Apache	2.4.7
TEXLive	3.1415926
JRE	1.8.0_66-b17
Apache Maven	3.3.3
Redmine	2.4.2
Jenkins	1.641

Table 2: Table of used Software and Version

B.3 Project Plan

This section covers the time used for this term project. The project is rewarded with 8 European Credit Transfer System (ECTS) points. For each point, 30 hours of work are estimated. This gives a total of 8 * 30 = 240 hours for the project. But since we had a bit of a problem with the beginning because it was not clear if we could do the project, we ended up running a week late, so this leads to a total of 13 weeks. This means we would have a workload of 240 / 13 = 18.5 hours/week.

B.3.1 Actual vs. Planned work hours

In Figure 36 the actual vs. the target hours/week are compared.

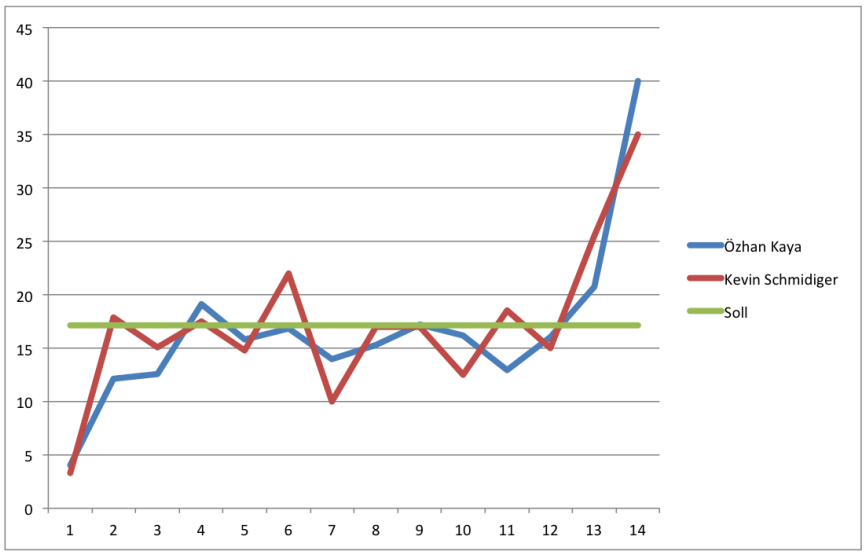

Figure 36: Actual vs. Target Hours/Week

In the beginning we haven't spend much time since we weren't sure if we can actually start as mentioned before. We knew at the end of the first week, that we had the okay from the school to get started with the term project. In week 7 Kevin Schmidiger got very sick so we both didn't work much on the project. And the last week is not finished yet, but we estimated the effort on the project to get it fully done and that we are satisfied with the result.

B.3.2 Hours spent per student

Together we spent 473 hours on this term project according to our time tracking and project management tool Redmine. Here's how the distribution looked like:

Student	Hours
Kevin Schmidiger	240
Özhan Kaya	233

Table 3: Hours spent per student

References

[CDT15] Eclipse CDT. Website of the Eclipse CDT. `https://eclipse.org/cdt/`, 2015. [Online; accessed 16-December-2015].

[Cev15] Cevelop. Website of the Cevelop IDE. `https://www.cevelop.com/`, 2015. [Online; accessed 16-December-2015].

[Cod15] CodeAndMe. Website of Code and me of Feature plugin. `http://codeandme.blogspot.ch/2012/12/tycho-build-4-building-features.html`, 2015. [Online; accessed 16-December-2015].

[Cpp15] CppReference. CppReference of type alias, alias template. `http://en.cppreference.com/w/cpp/language/type_alias`, 2015. [Online; accessed 16-December-2015].

[Ecl15a] Eclipse. Contributing Actions to the Eclipse Workbench. `http://www.eclipse.org/articles/article.php?file=Article-action-contribution/index.html`, 2015. [Online; accessed 16-December-2015].

[Ecl15b] Eclipse. Website of the eclipse mars site. `http://download.eclipse.org/releases/mars`, 2015. [Online; accessed 16-December-2015].

[Ecl15c] Eclipse. Website of the eclipse update site. `http://download.eclipse.org/eclipse/updates/4.5`, 2015. [Online; accessed 16-December-2015].

[Hya15] Hyamsudar. Website of the Hyamsudar. `http://download.eclipse.org/eclipse/updates/4.5`, 2015. [Online; accessed 16-December-2015].

[IFS15a] IFS. Download Page of the PASTA plugin. `https://www.cevelop.com/cdt-testing/development/`, 2015. [Online; accessed 16-December-2015].

[IFS15b] IFS. Main page of IFS. `https://www.ifs.hsr.ch`, 2015. [Online; accessed 16-December-2015].

[ISO14] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2014.

[Jen15] Jenkins. Website of the jenkins. https://wiki.jenkins-ci.org/display/JENKINS/Xvfb+Plugin, 2015. [Online; accessed 16-December-2015].

[Rep15] Repara. Website of the Repara cdt-testing document. http://repara-project.eu/wp-content/uploads/2014/04/ICT-609666-D4.1.pdf, 2015. [Online; accessed 16-December-2015].

[Som15a] Prof. Sommerlad. Wiki page of Student Proposals. http://wiki.hsr.ch/PeterSommerlad/StudentProjectProposals, 2015. [Online; accessed 16-December-2015].

[Som15b] Prof. Sommerlad. Wiki page of the term project of AliExtor. http://wiki.hsr.ch/PeterSommerlad/AliExtor, 2015. [Online; accessed 16-December-2015].

[Vog15a] Lars Vogel. Website of Lars Vogel of Extensions. http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html, 2015. [Online; accessed 16-December-2015].

[Vog15b] Lars Vogel. Website of Lars Vogel of Tycho. http://www.vogella.com/tutorials/EclipseTycho/article.html, 2015. [Online; accessed 16-December-2015].

[Xvf15] Xvfb. Website of the Xvfb server. http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml, 2015. [Online; accessed 16-December-2015].

[Yve11] YvesThrier. TurboMove, Move Refactorings for Eclipse CDT. http://eprints.hsr.ch/289/, 2011. [Online; accessed 16-December-2015].