

HSR – University of Applied Sciences Rapperswil
Institute for Software

Bachelor Thesis

namespactor

**CDT Namespace
Refactoring Plug-in**

**metriculator speed-up
CDT metric Plug-in**

Ueli Kunz, ukunz@hsr.ch
Julius Weder, jweder@hsr.ch

<http://sinv-56013.edu.hsr.ch>

Supervised by Prof. Dr. Luc Bläser

June 15, 2012

Abstract

This bachelor thesis consists of two sub parts. The main part is about developing a namespace refactoring tool called `namespactor`. The other part dedicates to the further development of the `metriculator` plug-in that was initiated in our semester thesis at the IFS [ifs12]. Both projects are plug-ins for the Eclipse C/C++ Development Tooling Platform (CDT, [CDT11]).

`namespactor`

C++ allows to introduce names into program scopes by way of "using directives" and "using declarations", such that symbols can be referred to without their qualified name. However, the manual changing of names and the switching between their qualified and unqualified representation is error prone and time consuming. Therefore, we have developed `namespactor`, a new automated C++ refactoring tool for names and namespaces for Eclipse CDT. Modern integrated development environments (IDE) commonly provide some refactoring features, such as the features of `namespactor`.

`namespactor` offers a set of common and effective namespace refactoring functions, such as switching between qualified and unqualified naming, introducing and moving "using declarations" or "using directives" and more. `namespactor` is realized with the Eclipse Language Toolkit (LTK), the API for integrating automated refactorings in Eclipse IDE.

There is currently no similar refactoring tool publicly available for the programming language C++. The following refactorings are implemented in `namespactor`:

- **Inline Using Directive:** Removes a using directive and qualifies the affected names.
- **Inline Using Declaration:** Removes a using declaration and qualifies the occurrences of the affected name.
- **Qualify an Unqualified Name:** Fully qualifies an unqualified name with all required names.
- **Extract Using Directive:** Introduces a using directive for a qualified name and removes the name qualifier(s) of the affected qualified names.
- **Extract Using Declaration:** Introduces a using declaration for a qualified name and removes the name qualifiers(s) on the occurrences of the affected qualified name.

metriculator speed-up

The further development of metriculator mainly aimed to improve the performance of the static code analysis. At the end of the semester thesis metriculator lacked in releasing allocated memory. Various design changes allowed us to improve the algorithm for calculating the metric values and to constantly release memory.

After the improvements, metriculator performs the analysis up to four times faster and the memory allocation is reduced by 75%. It is now possible to analyse projects with about one million physical source lines of code in less than one minute.

Management Summary

This chapter summarises the goals and outcomes of the namespaceator project and gives a preview on what might be possible in the future. The section "Illustrated Example" demonstrates the use of namespaceator based a sample scenario.

The last section summarises the results of the metriculator speed-up project.

Initial Situation

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour [Fow]. C++ allows to allows to semantically and logically group names. A name nested within such a group must be qualified if used from outside. Thus, working in an environment with nested names automatically means more typing effort. More work raises the chance of manually introduced errors to the code. namespaceator will automate frequently required steps when working with names in C++ by providing different kinds of refactorings. For now, there is no similar tool publicly available for the Eclipse C/C++ Tooling Platform (CDT, [cod11]) or any other C++ integrated development environment (IDE).

Procedure and Technologies

namespaceator is a plug-in for CDT, which is itself a plug-in for Eclipse. CDT is a well known platform to develop C/C++ software, which provides an infrastructure called LTK (Language Toolkit, [ltk06]) to create refactorings.

Given a name as input, a namespaceator refactoring changes all referencing names in the code base. The kind of changes applied depend on the chosen refactoring and input name.

A smaller part of namespaceator relies on the static code analysis framework Codan [cod11]. namespaceator analyses the source code in a background process and reports potential problems related to names. For each reported problem, namespaceator suggests to apply one or more refactorings that will solve the problem.

Results

namespaceator features the following refactorings:

- Inline Using Directive: Removes a using directive and qualifies the affected names.

-
- **Inline Using Declaration:** Removes a using declaration and qualifies the occurrences of the affected name.
 - **Qualify an Unqualified Name:** Fully qualifies an unqualified name with all required names.
 - **Extract Using Directive:** Introduces a using directive for a qualified name and removes the name qualifier(s) of the affected qualified names.
 - **Extract Using Declaration:** Introduces a using declaration for a qualified name and removes the name qualifiers(s) on the occurrences of the affected qualified name.

namespacepactor is available as Eclipse plug-in and can be installed from within Eclipse using the install wizard. After installation, the namespacepactor refactorings are available within the Eclipse refactoring window menu. The Codan checkers of namespacepactor run as background processes and continuously analyse the source code for potential programming problems. Reported problems can automatically be solved by executing one of the suggested problem resolutions, which will start one of the namespacepactor refactorings.

Illustrated Example

This section outlines the usage of namespacepactor based on an illustrative scenario.

The example code in figure 0.1 illustrates how even in a small piece of code bad things can happen. This code prints a sentence to the console output and uses a library called "answers.h" to calculate an answer.

First of all, it is bad practice to write using directives before `#include` directives [Sut02]. This could lead to side effects in the included header file. It is also best practice to keep using directives as local as possible, which definitely the global scope is not.

namespacepactor detects the using directive that is placed before the last `#include` directive and reports a problem. The suggested quick fix is either to move it after the `#include` directive or to inline it.

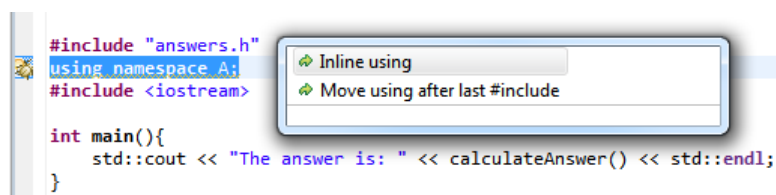


Figure 0.1.: Example of bad source code.

Applying the quick fix "Move using after last `#include`" moves the using directive after the last `#include` directive. Applying the quick fix "Inline using" initiates the inline using directive refactoring that removes the using directive and qualifies the affected function call "calculateAnswer()" with the name "A" as illustrated in figure 0.2.

```
#include "answers.h"
#include <iostream>

int main(){
    std::cout << "The answer is: " << A::calculateAnswer() << std::endl;
}
```

Figure 0.2.: Source code after the inline using directive refactoring was performed. The refactoring was initiated by a quick fix.

In this situation it is maybe desirable to introduce the namespace "A" inside the function "main". This namespace provides other useful functions and I do not want to always write the qualified name to access them. Selecting the function call "A::calculateAnswer()" as illustrated in figure 0.2, allows me to apply the extract using directive refactoring via the corresponding Eclipse refactoring window menu. This refactoring results in the source code illustrated in figure 0.3.

```
#include "answers.h"
#include <iostream>

int main(){
    using namespace A;
    std::cout << "The answer is: " << calculateAnswer() << std::endl;
}
```

Figure 0.3.: Source code after the extract using directive refactoring was performed. The refactoring was initiated by the corresponding Eclipse refactoring window menu.

The extracted using directive now affects only names in the scope of the function main. Hence, inside the function "main", it is still possible to benefit from the introduced namespace "A".

The result of steps shown above is a clean and unproblematic source code, achieved by using the simple and intuitive features of namespactor.

Future Work

For now, namespactor is a useful and powerful plug-in that accelerates and simplifies working with names. There are still some issues left open. Since namespactor was developed only within eleven weeks, it could still profit from further improvements. It would also help to get some feedback from experienced C++ developers to ensure that this plug-in can prove itself in real world projects.

Metriculator Speed-up

At the end of the preceding semester thesis [met11b] metriculator did not perform well on projects with more than about 300'000 physical source lines of code (PSLOC). At some point metriculator run out of memory and crashed.

The aim was to improve the performance, enabling metriculator to analyse 1 mio. PSLOC in less than 3 minutes.

The performance improvements gained during this bachelor thesis are well observable in figure 0.4. metriculator now runs up to four times faster than before, by acquiring only 75% of the amount of memory. The analysis of 1 mio. PSLOC is processed in less than one minute, compared to the initial situation where metriculator run out of memory and finally crashed.

metriculator was already published at the Eclipse marketplace and was downloaded over 40 times in the first month since the release [met12]. It performs now very well, even with large projects. Further metrics would increase the value of metriculator, see the metriculator documentation [met11b] to read more about interesting metrics.

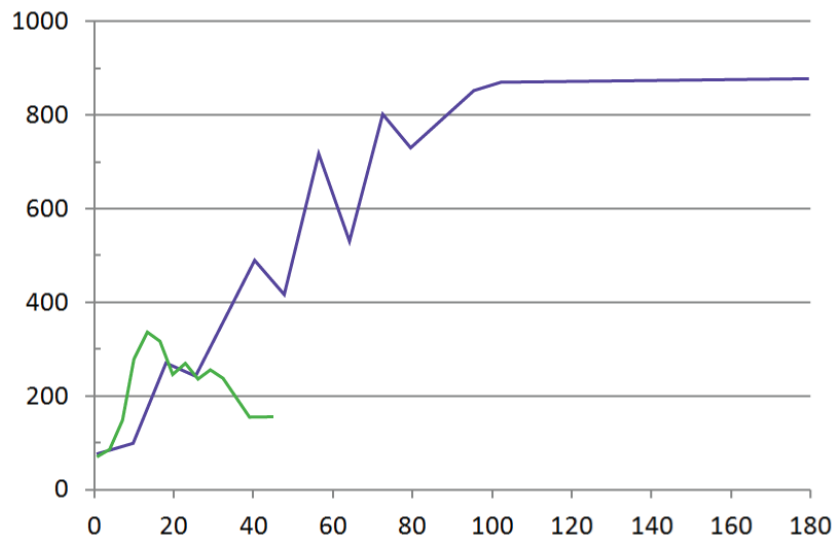


Figure 0.4.: Memory allocation before (blue) and after (green) the performance improvements analysing 1 mio. PSLOC. Horizontal axes: time in seconds. Vertical axes: allocated memory in mega bytes.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Project Duration	1
1.3. About this Document	1
2. Objectives	3
2.1. Common	3
2.2. namespace	3
2.3. metriculator	8
2.4. Agreement	9
3. Namespace	10
3.1. Analysis	10
3.1.1. User Stories - Inline Refactoring	10
3.1.2. User Stories - Extract Refactoring	11
3.1.3. Inline Refactorings	11
3.1.4. Extract Refactoring	22
3.2. Namespace Refactorings	31
3.2.1. Inline Using Directive (IUDIR)	31
3.2.2. Inline Using Declaration (IUDEC)	31
3.2.3. Qualify an Unqualified Name (QUN)	32
3.2.4. Extract Using Directive Refactoring (EUDIR)	32
3.2.5. Extract Using Declaration Refactoring (EUDEC)	33
3.3. Implementation	33
3.3.1. Plug-in Architecture	34
3.3.2. Refactoring User Interface - Language Toolkit (LTK)	35
3.3.3. Static Code Analysis with Codan	36
3.3.4. Building Names	39
3.3.5. Name Lookup Algorithm	40
3.3.6. Inline Refactorings	43
3.3.7. Extract Refactorings	51
3.4. Open Issues	59
3.4.1. Qualify Names Defined Outside of the Workspace - #273	59
3.4.2. Nested Using Directives within Namespace Definitions - #269	59
3.4.3. Finding Implicit Operator Calls - #270	59
3.4.4. Qualifying Template Method Definitions - #271	60
3.4.5. Missing Line Break after last Affected Node - #238	60
3.4.6. Creating Fully Qualified Names - #249	61
3.4.7. Using Declaration with Generic Template Argument - #239	61
3.4.8. Inherited Type Name cannot be Replaced - #231	62

3.5.	Future Improvements	63
3.5.1.	Implement Hybrid Lookup in IUDEC and QUN	63
3.5.2.	Detect Name Conflicts	63
3.5.3.	Start IUDIR on Any Name	63
3.5.4.	Extract Using Declaration Into a Type Declaration - #265	63
3.5.5.	Extension for the Refactoring Qualify an Unqualified Name - #265	65
3.6.	Conclusion	66
4.	Metriculator	67
4.1.	Requirements	67
4.1.1.	Performance	67
4.1.2.	Tag Cloud - Dealing with Large Data Input	68
4.1.3.	Composite Update Site	68
4.2.	Performance	68
4.2.1.	Performance Measurement - Comparison Before and After the Im- provements	68
4.2.2.	Performance Improvements	72
4.2.3.	Open Issues	74
4.3.	Tag Cloud - Dealing with Large Data Input	75
4.4.	Composite Update Site	75
4.5.	Design Changes	76
4.5.1.	Tag Cloud Extraction	76
4.5.2.	NodeInfo Refactoring	77
4.6.	Further Improvements	83
4.6.1.	GUI Guidelines	83
4.6.2.	Minor Bug Fixing	84
4.7.	Unit Testing	84
4.7.1.	Codan Test Infrastructure	85
4.7.2.	Checker Tests	85
4.7.3.	Indexer Based Tests	86
A.	Environment Set up	89
A.1.	Hardware	89
A.2.	Project Management Software	89
A.3.	Version Control System, Git	89
A.4.	Development Environment	89
A.5.	Build and Deployment Automation	90
A.5.1.	Maven XML Configuration	90
A.6.	Testing Eclipse CDT Refactoring Plug-ins	90
A.6.1.	CDTTesting Framework	90
A.7.	AST Rewrite Store	94
A.8.	DOM AST View	94
B.	Terminology	96
C.	CDTTesting Plug-in Set up	97
C.1.	Quick Start	97
C.2.	Set up for Refactoring Tests	97

D. IUDIR Refactoring - Indexer Implementation	99
D.1. Finding References Recursively	101
D.2. Open Issues	103
D.2.1. Qualification of Implicit Operator Call	103
E. User Manual	106
E.1. Example of a Refactoring - Inline Using Directive	106
E.2. Refactorings in namespactor	107
E.3. Run a Refactoring	107
E.4. Quick Fixes	108
E.4.1. Problem Resolutions (Quick Fixes)	109
F. Project Management	110
F.1. Project Plan	110
F.2. Time Schedules	111
F.3. Personal Impression	112
F.3.1. Ueli Kunz	112
F.3.2. Julius Weder	113
G. Nomenclature	118

You don't understand anything until you learn it more than one way.

Marvin Minsky

1. Introduction

This project consists of two sub projects. One is the further development of the metriculator plug-in [met11a]. The other sub project is the main task of this thesis. The topic is about developing a namespace refactoring tool for the programming language C++. The tool integrates as plug-in into the Eclipse CDT (C/C++ Development Tooling) platform [CDT11].

Refactoring is a process in software development to increase the quality of source code without changing its functionality. Refactoring facilitate certain steps in the process of creating or maintaining source code. It reduces the complexity of the source code and therefore the need of time consuming error analysis [Ref11].

Many IDEs (integrated development environment) provide tools to automate refactoring steps. Appropriately using these tools allows to efficiently apply refactorings and gain high quality source code. Refactoring tools also eliminate the risk of a fault introduced by human doing the same change manually many times.

There is no comparable namespace refactoring tool available for Eclipse CDT or any other C++ IDE.

1.1. Motivation

We intend to improve the quality of the Eclipse CDT platform as it will help other developers to create better software. Our namespace refactoring tool helps to deal with namespace directives and declarations. We give our best to implement a highly useful refactoring that is easy to use and integrates well into the Eclipse CDT.

1.2. Project Duration

This bachelor thesis starts on February 20th and has to be finished until June 15th, 2012.

1.3. About this Document

Since this bachelor thesis covers two sub projects, namespactor and metriculator, each of it has its own chapter in this document. The chapters are ordered by their weight. Although we started working on the metriculator project, the namespactor chapter 3

precedes the metriculator chapter 4. The Appendix and Conclusion as well as the Management Summary, Introduction and Objectives apply to both sub projects and are not part of the division.

Regarding heading capitalisation rules, we follow the MLA style [mla08].

2. Objectives

This chapter defines the topic of this bachelor thesis. The following definition of the objectives is divided into three sub sections. The first sub section commonly applies to this thesis. The second and third sub section define goals for each of the two sub projects.

2.1. Common

Project Organisation The project is organised in fixed one-week iterations. Redmine [red11] is used for planning, time tracking, issue tracking and as information radiator for the advisor and supervisor. A project documentation is written during the project. Organisation and results are reviewed weekly together with the advisor and supervisor.

Integration and Automation Sitting in front of a fresh Eclipse CDT installation a first semester student can install our namespace refactoring plug-in. An update site is created to allow the installation of namespactor using the Eclipse install wizard as long as namespactor is not integrated into the CDT plug-in.

Quality The plug-in code is covered with automated test cases. Automated UI tests are not mandatory. Automated tests are created for the commands triggered by the UI.

Delivered Assets At the end, the project will be handed to the supervisor with two CDs and two paper versions of the documentation. The CDs contain: this project report, a poster explaining namespactor, the source codes of both plug-ins and the deployable plug-ins.

2.2. namespactor

namespactor is a namespace refactoring tool for the programming language C++. The tool is available as plug-in for the Eclipse CDT platform. The main purpose is to provide a tool that automates inline and extract operations on names. Such operations are often repetitive and may lead to faulty code if done by hand.

The supported operations are explained in detail in the following descriptions:

Inline Namespace This functionality inlines name specifier(s) referenced in a using directive. The using directive will be removed and the affected unqualified names in the source code will be expanded by the name specifier(s) of the removed using directive as illustrated in listing 2.1.


```
namespace A{
    int a();
}

/* before the inline refactoring */

void doIt(){
    using namespace A;
    a();
}

/* after the inline refactoring */

void doIt(){
    A::a();
}
```

Listing 2.1: Code snippet illustrating the inline refactoring with a using directive.

This functionality can also be achieved with using declarations as illustrated in listing 2.2.

```
namespace A{
    int a();
}

/* before the inline refactoring */

void doIt(){
    using A::a;
    a();
}

/* after the inline refactoring */

void doIt(){
    A::a();
}
```

Listing 2.2: Code snippet illustrating the inline refactoring with a using declaration.

Extract Namespace This functionality removes occurrences of name specifier(s) in the source code and introduces the required namespace with a using directive as illustrated in listing 2.3.

```
namespace A{
    int a();
}

/* before the extract refactoring */

void doIt(){
    A::a();
}

/* after the extract refactoring */

void doIt(){
    using namespace A;
    a();
}
```

Listing 2.3: Code snippet illustrating the extract refactoring with a using directive.

This functionality can also be achieved with using declarations, by introducing the qualified name with a using declaration as illustrated in listing 2.4.

```
namespace A{
    int a();
}

/* before the extract refactoring */

void doIt(){
    A::a();
}

/* after the extract refactoring */

void doIt(){
    using A::a;
    a();
}
```

Listing 2.4: Code snippet illustrating the extract refactoring with a using declaration.

Remove Unused Using (Optional) Using directives/declarations that are not required in the source code will be removed as illustrated in listing 2.5.

```
/* begin of file */
#include <iostream>
using namespace std;

namespace A{
    int a();
}

int main(){
    using namespace A;          // never used
    cout << "Hello" << endl;
    return 0;
}
/* end of file */
```

Listing 2.5: Code snippet illustrating the meaning of an unused using.

Remove Needless Nested-Name-Specifier (Optional) Removes name specifier(s) if they are redundant because the symbol is already available through a using directive/declaration. Listing 2.6 illustrates a case where needless name specifier(s) are used.

```
using namespace A;
cout << hello() << endl;
cout << A::hello() << endl;  // needless qualified name

cout << B::hello2() << endl; // ok
```

Listing 2.6: Code snippet using a needless name specifier.

Force Qualified Name in a Using Directive (Optional) Make unqualified-ids in using directives qualified. If nested using directives are spread over a file, it could be difficult to see which namespace is nested in an other namespace, especially in big source files. The refactoring also helps to prevent name conflicts.

Listing 2.7 shows an example with qualified names in using directives. Further details are shown in the refactoring analysis in section 3.1.

```
namespace A {  
    namespace B {  
        namespace C {  
        }  
    }  
}  
  
using namespace A;  
using namespace B;           // before refactoring  
//using namespace A::B;     // after refactoring  
using namespace C;           // before refactoring  
//using namespace A::B::C;  // after refactoring
```

Listing 2.7: Code snippet illustrating the effect of the force qualified name in a using directive refactoring.

2.3. metriculator

At the end of the semester thesis, in which metriculator was initiated, some issues were left open to be fixed in future releases. In this bachelor thesis, we address the most important open issues.

Bug Fixing The issues listed below are referenced with the issue number of our Redmine project [met11a]:

- Composite update site not working (#177)
- Logical merging of function definitions and declarations within anonymous namespaces does not work (#166)
- Tag cloud throws exception on big input (#176)

Increase Performance metriculator does still not perform very well when analysing large projects. Large means more than 200'000 LSLOC (Logical Source Lines of Code). We are going to address this issue in this bachelor thesis with the aim to make metriculator also a valuable tool for large projects. See the requirements section 4.1.1 where we define how the performance is measured.

Commit Plug-in to CDT We are strongly encouraged to make metriculator officially available to other developers. Therefore, we introduce metriculator to the CDT community with the aim to commit it to an upcoming release.

Further Metric (Optional) To make metriculator even more valuable, we plan to implement further metrics.

- **Number of Usings in Header Files:** Measures the number of using directives in header files. Since placing using directives in header files is bad practice [Sut00], metriculator may suggest a quickfix to inline the using directives with the inline refactoring of namespace, see section 2.2.
- **Number of Function Calls using ADL:** If the name of a function cannot be resolved in its lexical context, the compiler tries to find the function in the namespaces of its arguments [Str09, paragraph 8.2.6]. This feature is known as ADL (argument-dependent lookup or argument-dependent name lookup). Reading source code with unqualified function calls may be confusing and lead to semantic problems [Sut]. Therefore, metriculator may suggest a quickfix to qualify the unqualified name of the function call. The refactoring is illustrated in Listing 2.8.

```
namespace Time {
    class Date{};
    std::string format(const Date&);
}

void f(Time::Date d, int i){
    /* before refactoring - refactoring candidate is format(d) */
    std::string s = format(d); // implicitly resolved to Time::format()
    std::string t = format(i); // Error: format cannot be resolved

    /* after refactoring */
    std::string s = Time::format(d); // explicitly call Time::format()
    std::string t = format(i); // Error: format cannot be resolved
}
```

Listing 2.8: Code snippet showing an implicitly resolved name of a function call with ADL and the result after the inline refactoring was applied.

2.4. Agreement

The following contracting parties agree upon the objectives of this bachelor thesis described in the Objectives chapter (2).

Place, Date:

Dr. Prof. Luc Bläser, Signature:

Ueli Kunz, Signature:

Julius Weder, Signature:

3. Namespactor

This chapter contains information that relate to the second phase but the main task of this thesis. This is the development of a namespace refactoring plug-in called namespactor. See section 2.2 in chapter Objectives for a brief explanation of the purpose and motivation behind namespactor.

This chapter starts with the analysis section 3.1 of the inline and extract refactorings. Section 3.2 defines the refactorings based on the findings during the analysis. The high and low level software design of the namespactor plug-in is outlined in section 3.3.1 followed by section 3.3 which describes the implementation details.

Appendix B defines the terminology for this chapter.

3.1. Analysis

This section examines namespactor from different perspective. Starting from a users perspective we define the user stories. Based on that, the inline and extract refactorings, introduced in chapter 2.2, are investigated in detail. We look at different approaches on how to apply the refactorings regarding to various aspects such as nesting levels and scope of application.

3.1.1. User Stories - Inline Refactoring

A user can trigger the inline refactoring from different types of locations in the source code. He may want to inline a using directive or using declaration, or he may want to qualify an unqualified name. This leads to three sub types of the inline refactoring.

Remove a Using Directive A user wants to remove a using directive. The user marks the using directive and runs the inline refactoring of namespactor. The refactoring expands the affected unqualified names in the source code with the required name specifier(s).

Remove a Using Declaration A user wants to remove a using declaration. The user marks the using declaration and runs the inline refactoring of namespactor. The refactoring expands the affected unqualified names in the source code with the required name specifier(s).

Qualify an Unqualified Name A user wants that an unqualified name gets qualified. The user marks the unqualified name to be qualified and runs the inline refactoring of namespactor. The refactoring expands the affected unqualified name with the required name specifier(s).

3.1.2. User Stories - Extract Refactoring

A user can trigger the extract refactoring from different types of locations in the source code. He may want to introduce a using directive or a using declaration by selecting a qualified name. This leads to two sub types of the extract refactoring.

Introduce a Using Directive A user wants to introduce a namespace of a qualified name with a using directive. The user marks the qualified name and runs the extract refactoring of namespactor. The refactoring extracts the name specifier(s) into a using directive.

Introduce a Using Declaration A user wants to introduce a local synonym of a qualified name with a using declaration. The user marks the qualified name and runs the extract refactoring of namespactor. The refactoring extracts the name-specifier(s) into a using declaration.

3.1.3. Inline Refactorings

This section describes the purposes and examines the aspects of the inline refactorings.

3.1.3.1. Inline Using Directive

A using directive introduces names from a namespace, so the names are available without qualification. Using directives in the global namespace should better be avoided, it is rather a tool for a transitional solution [Str09, paragraph 8.2.3]. Using directives facilitate the reading and writing of source code. Sometimes it would be helpful or necessary to remove a using directive, that was inserted during a transitional phase for instance. If a using directive is removed, the unqualified names affected by the using directive need to be qualified. This could be a lot of work, if it has to be done by hand. The inline refactoring automatically expands the affected names with the required name specifier(s).

As already mentioned, using directives in the global scope should be avoided. Therefore, the inline refactoring can be applied to remove a using directive. In a further refactoring step it could be desirable to introduce the previously inlined namespace or another namespace in a more local scope. To introduce a namespace, the extract refactoring can be applied 3.1.4.

3.1.3.2. Inline Using Declaration

With a using declaration it is possible to determine the scope of application for a name. The using declaration creates a local synonym. Usually it is a good idea to keep the local synonyms as local as possible to avoid confusion [Str09, paragraph 8.2.2]. To avoid this confusion, it could be helpful to remove a using declaration. If a using declaration is removed, all occurrences of the affected unqualified name need to be qualified. The inline refactoring automatically expands all occurrences of the unqualified name with the required name specifier(s).

If one or more namespaces are introduced with a using directive, a using declaration can be used to create a local synonym to prevent name conflicts, see extract refactoring 3.1.4 and listing 3.1 for more details. The using declaration reduces typing efforts and therefore typing errors. After the using declaration is in place, the inline refactoring can be applied to qualify all occurrences of the unqualified name, introduced by the using declaration. These two steps result in a source code without name conflicts.

3.1.3.3. Qualify an Unqualified Name

An unqualified name in a source code, that should be qualified because of clarity for instance, can be qualified with the inline refactoring. Qualifying a name is not just about simply qualifying this name, maybe it is also desired that all occurrences of other names from the same namespace are qualified. If all names from a namespace are qualified, the corresponding using directive could be removed or could be placed in another scope, for instance in another file of the translation unit [cpp11, paragraph 2.1.1].

The same applies to the using declarations as well, where all occurrences of a name are qualified so that the using declaration could be removed or could be placed in another scope.

The listings 3.1, 3.2, 3.3, 3.4 illustrate the different approaches of the inline refactoring by qualifying an unqualified name.

```
void doIt(){
    using namespace A;
    f1(); // apply inline refactoring here
    if(true){
        f2();
    }
    f3();
}
```

Listing 3.1: Foundation for the snippets 3.2, 3.3, 3.4 related to the inline refactoring by qualifying an unqualified name.

```
// 1st approach - only the chosen name is qualified
using namespace A;
A::f1();
if(true){
    f2();
}
f3();
```

Listing 3.2: First approach of the inline refactoring by qualifying an unqualified name.

```
// all occurrences of names from the same namespace are qualified
using namespace A;
A::f1();
if(true){
    A::f2();
}
A::f3();
```

Listing 3.3: Second approach of the inline refactoring by qualifying an unqualified name.

```
// same as 2nd approach + removing the "using namespace A" directive
A::f1();
if(true){
    A::f2();
}
A::f3();
}
```

Listing 3.4: Third approach of the inline refactoring by qualifying an unqualified name.

3.1.3.4. Examination of General Aspects

Following paragraphs describe aspects that apply generally to the inline refactorings.

3.1.3.4.1. Nested Namespaces Using nested namespaces is common for practical reasons as well as constructs can simply be nested where it makes sense [Sut02]. Namespaces are used to encapsulate names, in which case namespaces are preferred to classes. Encapsulating names can also help providing better readability.

There are different aspects to consider when applying the inline refactoring with respect to nested namespaces. The refactoring can be applied to the enclosing namespace or to a node that is in a nested namespace.

The listings 3.5, 3.6, 3.7 illustrate approaches of the inline refactoring applied to the enclosing namespace. The listing 3.8, 3.9, 3.10, 3.11 illustrate three approaches of the inline refactoring applied to the nested namespace.

3.1.3.4.2. Inline Name from an Enclosing Namespace Two approaches are shown. The first approach in listing 3.6 qualifies all names with the required namespace A. The second approach in listing 3.7 inlines the namespace A and its nested namespace B. The second approach could also be achieved by applying the inline refactoring as described in the first approach and then applying it again on the function b() or the using namespace A::B directive.

```
namespace A{
    int a();
    namespace B{
        int b();
    }
}

void doIt(){
    using namespace A;
    using namespace B;
    a(); // apply inline refactoring here
    b();
}
```

Listing 3.5: Foundation for the snippets 3.6, 3.7 illustrating the inline refactoring with respect to nested namespaces.

```
// 1st approach - inline namespace A
using namespace A::B; // qualify B with A
A::a();               // qualify a() with A
b();                  // b is known
```

Listing 3.6: First approach of the inline refactoring with respect to nested namespaces.

```
// 2nd approach - inline namespaces A and B
A::a();
A::B::b();
```

Listing 3.7: Third approach of the inline refactoring with respect to nested namespaces.

3.1.3.4.3. Inline Name from a Nested Namespace The first approach in listing 3.9 qualifies the function `b()` with the required namespace `B` and removes the using directive for the namespace `B`. The second approach in listing 3.10 inlines the namespace `B` and its enclosing namespace `A`. All names of these two namespaces are now qualified. The second approach can also be achieved by first applying the inline refactoring as described in the first approach and then by applying it again on function `b()` or function `a()`. The third approach in listing 3.11 qualifies function `b()` with the required namespaces and removes the using directive for namespace `B`.

```
namespace A{
    int a();
    namespace B{
        int b();
    }
}

void doIt(){
    using namespace A;
    using namespace B;
    a();
    b(); // apply inline refactoring here
}
```

Listing 3.8: Foundation for the snippets 3.9, 3.10, 3.11 illustrating the inline refactoring with respect to nested namespaces.

```
// 1st approach - inline namespace B
using namespace A;
a(); // a() is known
B::b(); // qualify b() with B
```

Listing 3.9: First approach of the inline refactoring with respect to nested namespaces.

```
// 2nd approach - inline namespaces A and B
A::a();
A::B::b();
```

Listing 3.10: Second approach of the inline refactoring with respect to nested namespaces.

```
// 3rd approach - qualify b()
using namespace A;
a();
A::B::b();
```

Listing 3.11: Third approach of the inline refactoring with respect to nested namespaces.

3.1.3.4.4. Nested Using Directives Using directives can be placed inside of a namespace definition. Therefore, if a namespace is introduced with a using directive, other namespaces may be introduced transitively.

The listings 3.12 and 3.13, 3.14 illustrate cases of using directives nested in namespace definitions with its impact to the inline refactoring.

Applying the inline refactoring to the function call `a()` qualifies it with `A`. If the inline refactoring would have been applied to the function call `b()`, the results looks very similar. The function call `b()` would have been qualified with `B`. Qualifying an unqualified name does not affect other names but the one the refactoring was initiated from.

But based on listing 3.12, we could also want to inline the using namespace `B` directive. In this case a transitive inline refactoring is required. Such a refactoring is required if the namespace to be inlined inlines another namespace. This is the case in listing 3.12, where namespace `B` has a using directive for the namespace `A`. This means that inside of the function `doIt()` all names from namespace `A` and `B` are known.

When inlining the using directive `B`, the directive is removed and the function call `b()` is qualified with `B`. Now, the function call `a()` cannot be resolved. When removing a using directive it is also necessary to recursively lookup for nested using directives inside of its target namespace. Therefore, the function call `a()` has to be qualified with `A`. The result of this transitive inline refactoring is shown in listing 3.14.

```
namespace A{
    int a();
}
namespace B{
    using namespace A;
    int b(){
        return a();
    }
}
void doIt(){
    using namespace B; // also introduces namespace A
    a(); // apply inline refactoring here
    b();
}
```

Listing 3.12: Foundation for the listing 3.13 illustrating the result of the inline refactoring with respect to nested using directives.

```
namespace A{
    int a(); // untouched
}
namespace B{
    using namespace A;
    int b(){
        return a();
    }
}
void doIt(){
    using namespace B;
    A::a(); // qualify a() with A
    b();
}
```

Listing 3.13: Result of the inline refactoring with respect to nested using directives.

```
namespace A{
    int a(); // untouched
}
namespace B{ // untouched
    using namespace A;
    int b(){
        return a();
    }
}
void doIt(){ // using namespace B removed
    A::a(); // qualify a() with A, transitive dependency
    B::b(); // qualify b() with B
}
```

Listing 3.14: Result of the inline refactoring with respect to nested using directives.

3.1.3.4.5. Multiple Inheritance In contrast to using directives, using declarations can be placed in composite type scopes (class or struct). This feature gets especially relevant, if inheritance is in place. A using declaration can then be used to avoid ambiguous names.

Listing 3.15 shows the foundation for a inline using declaration refactoring with multiple inheritance. It contains four structs, where the last struct (AB) inherits from two structs A and B. The struct A further inherits from struct U. In this example we examine the inlining of the using declaration 'using A::f' in struct AB.

The problem is that without this using declaration, the function call `ab.f(1)` in the main method would be ambiguous. Because, without the using declaration, the function call can not be resolved to an exact match (`f(int)`). Without the using declaration, the function `A::f(int)` is not visible from within the main method. Instead, the compiler tries to find other possible matches by implicitly converting the parameter. The implicit targets are `B::f(double)` and `AB::f(char)`. Since two targets are found the function call is ambiguous.

As a result, the inline using declaration refactoring has to qualify function calls that

reference a function that was introduced by the using declaration to be inlined (in this case 'using A::f'). The result of the refactoring is shown in listing 3.16.

```
struct U {
    int f(int i);
    char f(char c);
};
struct A : public U {
    int f(int i){}
    // shadow U::f(char)
    char f(char c);
};
struct B {
    double f(double d) {}
};
struct AB : public A, public B {
    // make A::f(int) and A::f(char) accessible from outside,
    // references are ab.f(1), ab.f('a');
    using A::f; // apply inline refactoring here
    // make B::f(double) accessible from outside
    using B::f;
    // hide A::f(char) (only if 'using A::f' in place)
    char f(char c){}
};
int main(){
    AB ab;
    // A::f(int) exact match, without 'using A::f' => ambiguous,
    // because multiple implicit conversions exist (B::f(double), AB::f(char)).
    // Declarations are: A::f(int) => qualify
    ab.f(1);
    // AB::f(char) exact match. Declarations are: AB::f(char), A::f(char)
    // => no need to qualify, since the definition AB::f(char) is called
    ab.f('a');
    // B::f(double) exact match, without 'using B::f' => ambiguous,
    // because multiple implicit conversions exist (A::f(int),
    // A::f(char), AB::f(char))
    ab.f(3.14);
}
```

Listing 3.15: Str.]Foundation to analyse multiple inheritance issues related to the inline using declaration refactoring. The code of this listing is based on [Str09, Page 419].

```
struct AB : public A, public B {
    // using declaration removed
    using B::f;
    char f(char c){}
};
int main(){
    AB ab;
    ab.A::f(1); // qualify f with A
    ab.f('a');
    ab.f(3.14);
}
```

Listing 3.16: Result of the inline using declaration refactoring based on listing 3.15.

3.1.3.4.6. Scopes An important aspect by applying the inline refactoring is in which scope the refactoring takes place. There are different approaches, as illustrated in the listings 3.17, 3.18, 3.19, 3.19.

The first approach qualifies all the names in the same scope including the inner scope(s). The second approach qualifies all the names in the same scope and pushes down the removed using directive into the inner scope(s). By applying this approach, it is possible that the using directive is introduced in several inner scopes. The third approach removes all the equivalent using directives in the chosen scope, in this case the file scope. Therefore, all names of this namespace have to be qualified. Other scopes available to choose could be file, translation unit, namespace, folder and project.

```
namespace A{
    int a();
}

void doIt(bool is){
    if(is){
        using namespace A; // apply inline refactoring here
        a();
        if(true){
            a();
        }
    }
    using namespace A;
    a();
}
```

Listing 3.17: Foundation for the listings 3.19, 3.19 illustrating the approaches of the inline refactoring with respect to the scope it applies to.

```
// 1st approach
void doIt(bool is){
    if(is){
        A::a(); // qualify here
        if(true){
            A::a(); // qualify here
        }
    }
    using namespace A;
    a();
}
```

Listing 3.18: Result of the inline refactoring with respect to the scopes it applies to.


```
// 2nd approach
void doIt(bool is){
    if(is){
        A::a(); // qualify here
        if(true){
            using namespace A;
            a();
        }
    }
    using namespace A;
    a();
}
```

Listing 3.19: Result of the inline refactoring with respect to the scopes it applies to.

```
// 3rd approach
void doIt(bool is){
    if(is){
        A::a(); // qualify here
        if(true){
            A::a(); // qualify here
        }
    }
    A::a(); // qualify here
}
```

Listing 3.20: Result of the inline refactoring with respect to the scopes it applies to.

Another special case, which was also mentioned as bad practice, is if there are using directives in an include file (*.h) [Sut02, Item 40]. If an implementation file (*.cpp) includes such a header file it implicitly introduces the names already introduced by using directive(s) in the header file. The inline refactoring could also have impacts to other files in the translation unit than the file where the refactoring was initiated from. A simple example is illustrated in the listings 3.21, 3.22, 3.23. The example works also for using declarations. Considering there is a using declaration instead of a using directive, the result of the inline refactoring is similar.

```
/* MyExample.h */
#include "Other.h"      // contains namespace A with a function a()
using namespace A;     // apply inline refactoring here (option 1)
int hello();           // normal function declaration

/* MyExample.cpp */
#include "MyExample.h"

void doIt(){
    hello();
    a(); // apply inline refactoring here (option 2)
}
```

Listing 3.21: Foundation code snippet to illustrate the impact of an inline refactoring across multiple files. Listings 3.22, 3.23 show the results. Considering there is a function declaration `a()` in the namespace `A` inside the `Other.h` header file, the refactoring qualifies the method call `a()` with the namespace name `A`. This applies for both options, 1 and 2. Option 1 additionally removes the `using namespace A` directive.

```
/* after refactoring option 1 */

/* MyExample.h */
#include "Other.h"
// using directive removed

/* MyExample.cpp */
void doIt(){
    hello();
    A::a(); // qualify here
}
```

Listing 3.22: Result of the inline refactoring over multiple files, option 1.

```
/* after refactoring option 2 */

/* MyExample.cpp */
void doIt(){
    hello();
    A::a(); // qualify here
}
```

Listing 3.23: Result of the inline refactoring over multiple files, option 2.

The inline refactoring could also be initiated from another file than illustrated in listing 3.21. The chosen scope of the refactoring could be the translation unit for instance. The refactoring has to look up in the whole translation unit which namespaces are introduced and based on it qualify the names.

Another approach is to move the using directive to the implementation file. It is also possible that other files include the header file with the using directive in it, even other header files. The aim could then be to move the using directive until it is no longer in a header file.

3.1.4. Extract Refactoring

This section describes the purposes and examines the aspects of the extract refactorings.

3.1.4.1. Introduce a Using Directive

Introducing names with a using directive often reduces typing effort but you have to pay attention with using directives. Introducing using directives can cause name conflicts. It is also bad practice to write using directives in header files or before include directives [Sut02, Item 40]. Therefore, introducing a using directive with the extract refactoring should be considered carefully, especially in which scope it takes place. To avoid name conflicts, it is probably the best to keep the using directives as local as possible.

3.1.4.2. Introduce a Using Declaration

Creating a local synonym with a using declaration, which brings in specific and selected names, often reduces typing effort. It is not like a using directive that introduces all names of a namespace. Using declarations follow the same guidelines as using directives, that is to keep the using declarations as local as possible and to not write them in header files or before include directives [Sut02, Item 40].

Another purpose to introduce a local synonym with a using declaration is to avoid name conflicts. If, for instance, two functions with the same signature are valid and its desirable to use always only one of them in a chosen scope. Listing 3.24 illustrates the effect of an extract refactoring that creates a using directive.

```
void doIt(){
    //before
    using namespace A;    // has a function f()
    using namespace B;    // has a function f()
    cout << f();          // f is ambiguous (name conflict), apply refactoring here

    //after
    using namespace A;
    using namespace B;
    using A::f;           // introduced using declaration to avoid name conflicts
    cout << f();          // equal to A::f()
}
```

Listing 3.24: Code snippet illustrating the priority of a using declaration over a using directive. It helps to prevent name conflicts. Consider both namespaces (A and B) have a function f().

3.1.4.3. Examination of Common Aspects

The following paragraphs describe aspects that generally apply to the extract refactorings.

3.1.4.3.1. Nested Namespaces There are different approaches how the extract refactoring is applied if nested namespaces are involved. These are illustrated in the listings 3.25, 3.26, 3.27, 3.28.

The first approach applies the refactoring on the foremost name specifier(s). It removes all occurrences of the chosen name specifier(s) to be removed and introduces the name(s) with the appropriate using directive. The second approach removes all occurrences of the name specifier(s) by introducing the name(s) with the appropriate using directive. The third approach is an improved variant of the second approach. It introduces the namespaces of each name specifier of the qualified name with the appropriate using directive and removes all occurrences of name specifier(s) that are no longer required.

```
namespace A{
    int a();
    namespace B{
        int b();
    }
}
void doIt(){
    A::B::b();    // apply extract refactoring here
```

Listing 3.25: Foundation for the listing 1,2,3 illustrating the extract refactoring with respect to nested namespaces and using directives.

```
// 1st approach
using namespace A;
B::b();
```

Listing 3.26: First approach of the extract refactoring with respect to nested namespaces and using directives.

```
// 2nd approach
using namespace A::B;
b();
```

Listing 3.27: Second approach of the extract refactoring with respect to nested namespaces and using directives.

```
// 3rd approach
using namespace A;
using namespace B;
b();
}
```

Listing 3.28: Third approach of the extract refactoring with respect to nested namespaces and using directives.

Applying the extract refactoring with a using declaration reveals a simpler approach than with using directives as illustrated in listing 3.29.

```
namespace A{
    int a();
    namespace B{
        int b();
    }
}
void doIt(){
    /* before the refactoring */
    A::B::b(); // apply extract refactoring here

    /* after the refactoring */
    using A::B::b;
    b();
}
```

Listing 3.29: Extract refactoring with respect to nested namespaces and using declarations.

3.1.4.3.2. Scope of Application An important aspect when applying the extract refactoring is where, or rather in which scope the refactoring takes place. There are different approaches as illustrated in the listings 3.30, 3.31, 3.32, 3.33.

The first approach introduces the namespace with a using directive and removes all occurrences of the name specifier(s) in the same scope (not in inner scopes). The second approach also introduces the namespace and removes all occurrences of the name specifier(s) in the same scope and its inner scopes. The third approach removes all occurrences of the name specifier(s) in the whole file.

```
namespace A{
    int a();
}
void doIt(){
    A::a();
    if(true){
        A::a(); // apply refactoring here
        while(true){
            A::a();
        }
    }else{
        A::a();
    }
}
```

Listing 3.30: Foundation for the listings 3.31, 3.32, 3.33 illustrating an extract refactoring with respect to the scope it applies to.

```
// 1st approach
A::a();
if(true){
    using namespace A;
    a(); // extract name here
    while(true){
        A::a();
    }
}else{
    A::a();
}
```

Listing 3.31: First approach of the extract refactoring with respect to the scope it applies to.

```
// 2nd approach
A::a();
if(true){
    using namespace A;
    a(); // extract name here
    while(true){
        a(); // extract name here
    }
}else{
    A::a();
}
```

Listing 3.32: Second approach of the extract refactoring with respect to the scope it applies to.

```
// 3rd approach
using namespace A;
a(); // extract name here
if(true){
    a();
    while(true){
        a(); // extract name here
    }
}else{
    a(); // extract name here
}
}
```

Listing 3.33: Third approach of the extract refactoring with respect to the scope it applies to.

3.1.4.3.3. Scope - Special Case Composite Type Neither using declarations nor using directives can be placed inside class or struct definitions, except using declaration nominating members of base types of the enclosing type declarations 3.1.4.3.4. This is especially important to be aware of, if an extract refactoring is applied on the type name of a member variable declaration. In such a case the using declaration or directive cannot be placed in the same scope as the refactoring was initiated from.

3.1.4.3.4. Scope - Inside Type Declaration In contrast to using directives, using declarations can be placed in composite type scopes (class or struct). This feature gets especially relevant, if (multiple) inheritance is in place. A using declaration can then be used to avoid ambiguous names.

It is possible to place a using declaration in the scope of a type declaration if the last name of a qualified name is a type member of a base type of the type declaration. This case is illustrated in listing 3.34. Listing 3.35 illustrates the result of the extract refactoring.

```
namespace A{
    struct S1{
        struct S1_1{};
    };
}
struct AB : A::S1{
    void f1(A::S1::S1_1); // apply extract refactoring here
};
```

Listing 3.34: Foundation of listing 3.35 illustrating the extract refactoring inside a type declaration. The type S1_1 is a member of the type S1 and S1 is a base type of the type AB which defines the scope of the refactoring.

```
namespace A{
    struct S1{
        struct S1_1{};
    };
}
struct AB : A::S1 {
    using A::S1::S1_1;
    void f1(S1_1);
};
```

Listing 3.35: Result of the extract refactoring inside a type declaration scope.

3.1.4.3.5. Scope - Name Hiding and Conflicts Another issue related to scopes is that extracting name qualifiers can lead to semantic errors. Listing 3.36 and 3.37 illustrate such a case.

```
namespace A{
    int var = 0;
}
int main(){
    int var = 0;
    A::var = 0;        // apply extract refactoring here
    return 0;
}
```

Listing 3.36: Foundation of listing 3.37 illustrating a name conflict introduced by an extract refactoring.

```
namespace A{
    int var = 0;
}
int main(){
    int var = 0;
    using namespace A;
    var = 0;           // removed A, binding changed to the declaration inside main
    return 0;
}
```

Listing 3.37: Result of an extract refactoring that changed the semantics of the source code. The binding of `var = 0` in the main function was changed from `A::var` to the `var` definition inside the main function.

Approaches about extracting namespaces into other files are not discussed since it does not lead to good design nor does it provide clarity.

3.1.4.3.6. Dealing with Name Conflicts Ambiguous names occur if one or more namespaces are introduced and if there are equivalent names. When applying the extract refactoring, introducing name conflicts should be avoided. There are different ways to deal with name conflicts:

- Cancel the extract refactoring.
- Apply the extract namespace and produce faulty source code and warnings.
- Let the affected name(s), otherwise conflicted names, qualified.
- Introduce a namespace alias and qualify the affected name(s) using the namespace alias.
- If names are in conflict, using declarations can resolve the conflicts.

3.1.4.3.7. Using Declaration Order Dependency A using declaration can only refer to names whose declarations have already been introduced before. A using directive, in contrast, brings in names introduced before and after the using directive. For further information about order dependencies see [Sut02, Item 40].

Therefore, it is important to find the correct place for the using declaration. The listings 3.38, 3.39, 3.40 illustrate the order dependency of the using declaration. Applying the extract refactoring on the function call `A::f(1)`, which currently calls `A::f(int)`, the using declaration has to be placed below the declaration of the function to be called. The using declaration has only seen the function declaration `f(double)`, if it is placed between the two namespace definitions of the namespace `A`. In this case, a silent implicit conversion from `int` to `double` occurs. Therefore, the using declaration must be placed after the declaration of the function `f(int)`. Either in the global scope or even better immediately before the function call. The latter would also follow the rule, stating that extractions must be placed as local as possible.

```
namespace A{
    int f(double i);
}

namespace A{
    int f(int i);
}

int main(){
    A::f(1); // calls f(int i); - apply extract refactoring here
    return 0;
}
```

Listing 3.38: Foundation for the listings 3.39, 3.40 illustrating approaches of an extract refactoring with respect to the order dependency of using declarations.

```
// 1st approach - misplaced using declaration
namespace A{
    int f(double);
}
using A::f;

namespace A{
    int f(int);
}

int main(){
    f(1); // calls f(double), uses a silent implicit conversion
    return 0;
}
```

Listing 3.39: First approach of the extract refactoring with respect to the order dependency of using declarations.

```
// 2nd approach - correctly positioned using declaration
namespace A{
    int f(double);
}

namespace A{
    int f(int);
}
using A::f; // first suitable place

int main(){
    using A::f; // second suitable place (as local as possible)
    f(1); // calls f(int);
    return 0;
}
```

Listing 3.40: Second approach of the extract refactoring with respect to the order dependency of using declarations.

3.1.4.3.8. Using Directive Order Dependency A using directive introduces names of a namespace that are declared before or after the directive. When extracting a namespace the using directive can be placed in various places. Each of which is discussed in this paragraph.

Same File, on Top This would mean that the namespace is also introduced to potentially afterwards included files. This can cause side effects in the included files which is not desirable [Sut02].

Same File, Right after Include Directives This solves the side effect problem from above. But since the file itself may be included by others this approach can lead to side effects as well.

Same File, with other Using Directives Placing the using directive right after the in-

clude directives may also have another issue. If other using directives, referring to outer namespaces of the one to be introduced, already exist right after the include directives, the position we insert our new using directive may be essential. But the position is only essential if the using directive does not use qualified names. Using fully qualified names in a using directive allows it to be placed independently to other using directives. See listings 3.41, 3.42, 3.43, 3.44 that illustrates the initial situation and possible refactoring solutions.

Same Scope, as Local as Possible Placing using directives as local as possible avoids side effects in other files (assuming that all names are encapsulated in namespaces). As described above the using directives may be order dependent if they do not use qualified names.

```
/* A.h */
namespace A{
    namespace B{
        void f();
    }
}

/* Impl.cpp */
#include "A.h";
using namespace A;
B::f(); // apply extract refactoring here
```

Listing 3.41: Foundation for the listings 3.42, 3.43, 3.44 illustrating an extract refactoring with respect to the order dependency of using directives.

```
// 1st approach - misplaced unqualified using directive
#include "A.h";
using namespace B; // error - B not known
using namespace A;
B::f();
```

Listing 3.42: First approach of the extract refactoring with respect to the order dependency of using directives.

```
// 2nd approach - correctly positioned unqualified using directive
#include "A.h";
using namespace A;
using namespace B;
B::f();
```

Listing 3.43: Second approach of the extract refactoring with respect to the order dependency of using directives.

```
// 3rd approach - correctly positioned qualified using directive
#include "A.h";
using namespace A::B; // possible place independent of other usings
using namespace A;
using namespace A::B; // possible place independent of other usings
B::f();
```

Listing 3.44: Third approach of the extract refactoring with respect to the order dependency of using directives.

3.2. Namespace Refactorings

This section defines the inline and extract refactorings based on the analysis results in section 3.1. Definitions that directly relate to an approach discussed in the analysis are annotated with the corresponding listing or section number.

3.2.1. Inline Using Directive (IUDIR)

Initial Condition Using directive in selection. (This differentiate from the analysis where the initial condition is a name, see section 3.1.3.4.)

Subject References of top level declarations in the namespaces to be inlined (this includes namespaces indirectly introduced by nested using directives) that are not qualified (3.14).

Motivation: Nested using directives are not modified. This could lead to side effects and is not directly related to inlining the selected using directive.

Scope Only references within the potential scope of the using directive (3.18).

Motivation: This is the most natural behaviour, since only code is affected that depends on the selected using directive.

Result Names are qualified with the name of the using directive, using directive removed (3.18, 3.22).

Motivation: The selected using directive is always removed because we think that is the natural behaviour the users expects, since the using directive has no effect after the refactoring.

Extensions • The user can choose the scope, e.g. file, project.

3.2.2. Inline Using Declaration (IUDEC)

Initial Condition Using declaration in selection. (This differentiate from the analysis where the initial condition is a name, see section 3.1.3.4.)

Subject References of the declaration nominated by the using declaration (3.15).

Motivation: This is the most natural behaviour, since only code is affected that

depends on the selected using declaration.

Scope References of the declarations nominated by the using declaration, that belong to one declaration, and therefore to one definition (3.15).

Motivation: References, that belong to more than one declaration, belong to a definition after the using declaration. Hence, they do not require qualification.

Result Names are qualified with the name used in the using declaration, using declaration removed.

Motivation: The selected using declaration is always removed because we think that is the natural behaviour the users expects, since the using declaration has no effect after the refactoring.

3.2.3. Qualify an Unqualified Name (QUN)

Initial Condition Unqualified name in selection. The name does not belong to a declarator.

Subject Selected name.

Scope Selected name (3.2).

Motivation: This is the behaviour the user expects. Qualifying other references of the same name may result in code changes not intended by the user.

Result Name is fully qualified.

Extensions • The user can choose the scope, e.g. file, project.

3.2.4. Extract Using Directive Refactoring (EUDIR)

Start Condition Qualified name in selection.

Subject References of top level declarations in the namespaces to be extracted. The using directive is extracted with all required name specifier(s) of the selected name (3.27).

Motivation: Using fully qualified names in a using directive allows it to be placed independently to other using directives (3.44). Using directives are introduced as local as possible.

Scope of Application Only references within the potential scope of the selected name (3.32).

Motivation: Keep the refactoring as local as possible.

Result Using directive with a qualified name above the first name to be changed (3.44). Other using directives with the same specifier(s) are also affected. The using directives are shortened with the name specifiers introduced by the extracted using directive. The shortening of a using directive can lead to the remove of the whole using directive.

Extensions • The user can choose the scope, e.g. file, project.

3.2.5. Extract Using Declaration Refactoring (EUDEC)

Start Condition Qualified name in selection.

Subject References of top level declarations in the declaration to be extracted. The using declaration is extracted with all required name specifier(s) of the selected name (3.29).

Scope of Application Only references within the potential scope of the selected name (3.32, 3.38).

Motivation: Keep the refactoring as local as possible.

Result Using declaration with a qualified name above the first name to be changed (3.44).

Extensions • The user can choose the scope, e.g. file, project.

3.3. Implementation

This section describes how namespactor is implemented. The C++ refactorings specified in 3.2 are implemented for the Eclipse CDT platform. The implementation is based on the CDT platform, the LTK (section 3.3.2) and the Codan framework (section 3.3.3, figure 3.1).

The sections 3.3.6 and 3.3.7 outline the details of the inline and extract refactoring the implementations.

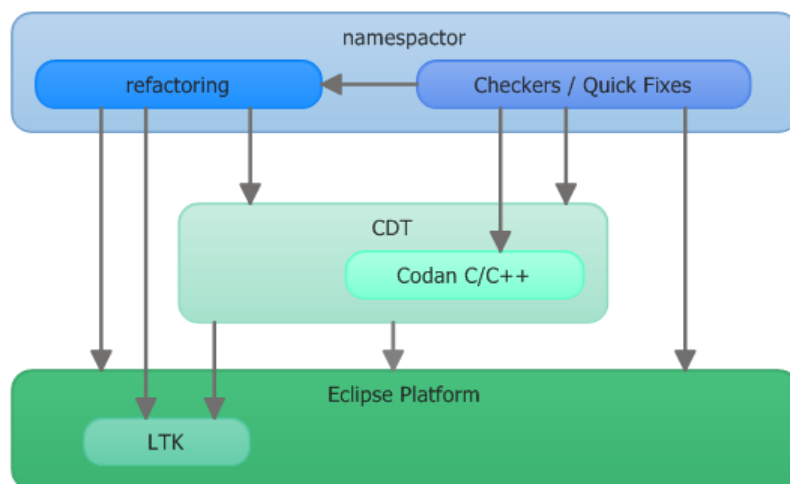


Figure 3.1.: Layer diagram with the dependencies of namespactor.

3.3.1. Plug-in Architecture

Figure 3.2 shows a dependency graph of the namespactor packages. Each refactoring has a ui and a refactoring package. The ui part is responsible to handle the Eclipse refactoring menu command by starting the associated refactoring. In contrast to the extract refactorings the inline refactorings feature a common ui package called iu (inline using). The inline refactorings have only one menu command. Depending on the selection either the IUDIR or the IUDEC refactoring is started. See 3.3.2 for more information about the user interface implementation.

The refactoring part is divided into a base package, which contains classes that are used by all refactoring implementations, and one package for each implemented refactoring. The refactoring base package contains the AST rewrite infrastructure (section A.7), an abstract base class for the refactorings and the TemplateIdFactory which is used to build template names (section 3.3.4.2).

The checkers package holds Codan checker implementations that report problems. The implemented resolutions in the quick fixes package start an appropriate refactoring to solve a reported problem. See section 3.3.3 for more information.

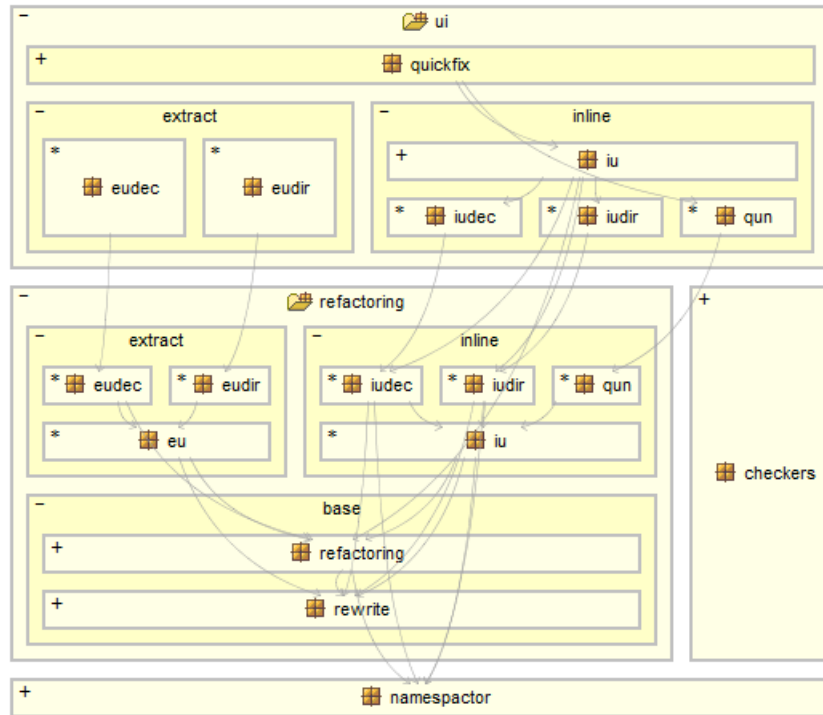


Figure 3.2.: Dependency graph of the namespactor core packages. The grey arrows show the dependencies between the packages. The `rtstest` (section A.6) and `astutils` packages are excluded from the diagram for convenience. The `extract`, `inline` and `base` frames were manually introduced to stress the belongings of the shown refactoring packages.

3.3.2. Refactoring User Interface - Language Toolkit (LTK)

To create automated refactorings for CDT, the language toolkit (LTK, [ltk06]) of Eclipse is used. A detailed description of the the LTK and its API can be found at [MI10] in chapter 2.1.

The LTK provides a wizard that allows to preview the changes of a refactoring (figure 3.3). Each of the refactorings has its own wizard implementation. Because we did not require to define any custom wizard pages, there was no need to customise the default implementation.

Along with the refactoring wizard other UI classes had to be implemented for every refactoring:

IWorkbenchWindowActionDelegate Required to bind an action to the UI via window menu or toolbar. Creates and invokes a refactoring action.

RefactoringAction Delegate between action delegate and refactoring runner. Creates a

RefactoringRunner2 and passes the selection of the active editor to it.

RefactoringRunner2 Instantiates the refactoring and passes this to a newly created refactoring wizard. The wizard is then passed to a helper class that starts the wizard if the initial conditions of the refactoring are met.

The implementation of these UI classes is very similar for every refactoring. The only exception is the implementation of the inline using (IU) refactoring UI classes. This is a special refactoring that, depending on the selection, invokes either the IUDIR or the IUDEC refactoring wizard. The dispatching is implemented in the IURefactoringRunner class. The advantage of the IU refactoring is, that only one inline refactoring entry is visible in the refactoring menu, see figure E.1.

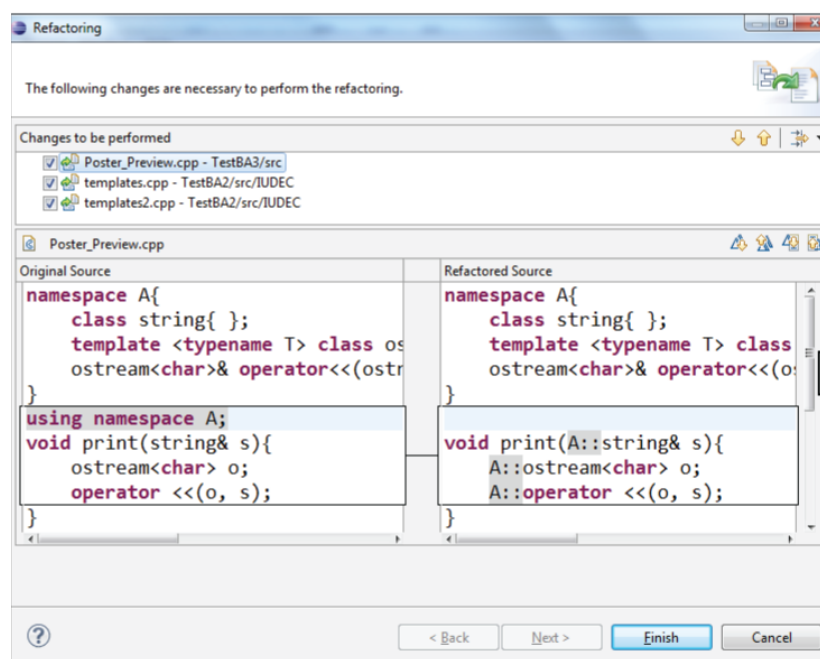


Figure 3.3.: Refactoring wizard showing the preview of an IUDIR refactoring.

3.3.3. Static Code Analysis with Codan

Codan is a static code analysis framework built upon Eclipse CDT [cod11]. In Codan, checkers are used to detect problems (e.g. dead code). Problems can be reported to Eclipse, which is able to visualise them in different ways, for example as markers in text editors. Optionally a problem marker has one or more resolutions. This is a suggestion to fix the marked problem (e.g. remove dead code branch). A detailed description of Codan and checkers can be found at [met11b] in chapter 4.3.

Checkers and problems have to be registered in the plugin.xml using an extension point. Another extension point allows to register resolutions for registered problems. See listing 3.45 featuring a sample entry that registers one checker with one problem and two resolutions for that problem.

```
<extension point="org.eclipse.cdt.codan.core.checkers">
  <checker class="namespactor.checkers.UsingChecker"
    id="ch.hsr.ifs.cdt.namespactor.checkers.UsingPosition">
    <problem category="org.eclipse.cdt.codan.core.categories.ProgrammingProblems"
      id="ch.hsr.ifs.cdt.namespactor.UDIRBeforeInclude">
    </problem>
  </checker>
</extension>
<extension point="org.eclipse.cdt.codan.ui.codanMarkerResolution">
  <resolution class="namespactor.quickfix.InlineUsingQuickFix"
    problemId="ch.hsr.ifs.cdt.namespactor.UDIRBeforeInclude">
  </resolution>
  <resolution class="namespactor.quickfix.MoveAfterIncludesQuickFix"
    problemId="ch.hsr.ifs.cdt.namespactor.UDIRBeforeInclude">
  </resolution>
</extension>
```

Listing 3.45: Code snippet from the plugin.xml registering one checker with one problem and two resolutions (quickfixes). Class names shortened.

The resolutions are implemented by extending one of these classes provided by Codan:

AbstractCodanCMarkerResolution Base class for all resolutions of problems reported with Codan.

AbstractAstRewriteQuickFix Extends AbstractCodanCMarkerResolution. Applies AST modifications to the document of the marker via template method.

3.3.3.1. Problems Detected by namespactor

namespactor features one checker that looks out for five problems, which are listed below. Except the last one, these problems are all based on the namespace guidelines documented by Herb Sutter in [Sut02] page 236. Each problem has one or more resolution as described in section E.4.1.

The implementation of this checker was not mandatory based on the objectives section 2.2, some of them were not even thought of at that point. But this checker generates a great surplus for namespactor, because the problem markers serve as good starting points to initiate one of the namespactor refactorings from. See figure 3.4 that suggests multiple quick fixes to solve a reported problem.

All the problems are detected on type. There is no need to explicitly run Codan to check for problems, this is done on the fly while typing. The following problems are reported by the namespactor checker:

Using Directive in Header File (UDIRInHeader) [Sut02] rule #1. A using directive in a header file, at the global scope, may lead to side effects. It introduces many names at the global scope. This may lead to unintended name conflicts in files that include the header file.

Using Declaration in Header File (UDECIInHeader) [Sut02] rule #2. A using (namespace) declaration in a header file has similar side effects as a using directive has.

But with using declarations this goes one step further. A using declaration should not be placed in a header file at all, because a using declaration may change the meaning of code depending on what `#include`s are in place ([Sut02] Example 40-3(c)).

Using Directive before `#include` (UDIRBeforeInclude) [Sut02] rule #3. Using directives before an `#include` may have side effects and change the semantics of the included files by introducing unexpected names.

Using Declaration before `#include` (UDECBeforeInclude) [Sut02] rule #3. Same as above.

Unqualified Name in Using Directive (QUNUDIR) An unqualified name in a using directive may decrease the readability of the code because it is not obvious where the name comes from. This problem is listed in the objectives section 2.2 as optional feature.

3.3.3.2. Problem Resolutions (Quick Fixes)

All of the reported problems can be solved by invoking one of the namespactor refactorings. Some problems have multiple resolutions and vice versa.

Inline Using (IU) Invokes the IU refactoring. Solves the problems: UDIRInHeader, UDECInHeader, UDIRBeforeInclude, UDECBeforeInclude.

Move Using After Include (MoveAfterIncludes) Moves the problematic using statement after the last `#include` directive. Solves the problems: UDIRBeforeInclude, UDECBeforeInclude. This is not a refactoring, since it may have side effects.

Qualify Unqualified Name (QUN) Invokes the QUN refactoring. Solves the problems: QUNUDIR.

The problem resolution implementations that invoke one of the namespactor refactorings extend the class `AbstractCodanCMarkerResolution`. The implementations simply instantiate the appropriate refactoring action and run it. The required information to run the action (active editor, problematic `ASTNode`) are determined by the data provided by the super classes.

The `MoveAfterIncludes` resolution implementation in contrast extends the `AbstractASTRewriteQuickFix` class. The AST rewrite is done in this implementation without delegation to a namespactor refactoring.

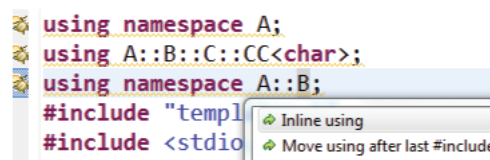


Figure 3.4.: Problem marker with suggested resolutions.

3.3.4. Building Names

namespactor has to deal with names. Either qualifiers are added (inline refactorings) or removed (extract refactorings) to a name or a name has to be built from scratch (names of using statements). In the AST, a name is represented by `IASTName` nodes. If a name is qualified, a specialised name, the `ICPPASTQualifiedName` is required. The `ICPPASTQualifiedName` contains multiple `IASTName` children. An unqualified name is represented by one `IASTName` instance or an `ICPPASTQualifiedName` with one `IASTName` child node. See figure 3.5 on the right that illustrates this.

A template name goes one step further, because names are nested. A template name contains an `IASTTemplateId` instance, which itself is a specialisation of `IASTName`. Figure 3.5 on the left, shows the AST of a nested template name.

An `IASTTemplateId` contains an `IASTName`, which is the template name, and an `ICPPASTTypeId`, that describes the node inside the angle brackets of the template name. An `ICPPASTTypeId` contains an `ICPPASTNamedTypeSpecifier` or an `ICPPASTSimpleDeclSpecifier`. The `ICPPASTSimpleDeclSpecifier` is a leaf node and represents a primitive type. The `ICPPASTNamedTypeSpecifier` contains one `IASTName` child, for example an `IASTTemplateId`, if so, the nesting of template names begins.

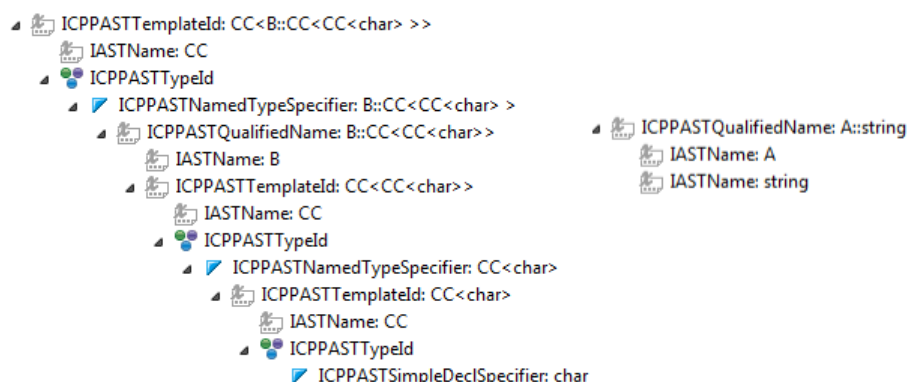


Figure 3.5.: AST of a template name with a qualified template argument name and an unqualified nested template argument name (left) and a qualified name (right).

The following sections describe how namespactor handles names.

3.3.4.1. Default Approach for non-Template Names

If a name without a template id has to be changed, the unchanged parts can just be copied and be reused in the new name. The names have to be copied because otherwise they are marked as frozen and can therefore not be inserted at another location. Every time the parser finishes its work, all nodes in the newly built AST are set to be frozen. Frozen nodes can not be changed. To modify a node retrieved from the AST, a copy of it is required.

For instance, if the name `B::C` has to be qualified with the name `A`, a new `ICPPASTQual-`

ifiedName is created, a new name node A is added first, followed by copies of B and C.

3.3.4.2. Template Names

The default approach, described above, does not work for template names for two reasons:

- First, copying template names does not work as expected. Nested template arguments are not written by the CDT ASTRewrite.
- Second, since template names contain nested names, it is possible that multiple names within a template name instance require to be changed. Nested names are problematic, because rewrite changes that affected nested nodes can not be handled by the ASTRewrite.

For instance, the template name `B::C::S<B::C::X>` requires to be qualified with A at two locations (given that B is within A).

Once at the outer template name `B::C::S` and once at the template argument `B::C::X` resulting in `A::B::C::S<A::B::C::X>`.

The solution we implemented to change template names is described in the following paragraph.

3.3.4.2.1. Changing Template Names To change template names, namespactor features an abstract factory base class called `TemplateIdFactory`. Given an existing `IASTTemplateId` as input, the factory visits each node in the `IASTTemplateId` and creates a new `IASTName` node step by step. Different specialisations of the base class implement different name changing mechanisms. Following specialisations exist:

CopyTemplateIdFactory This factory does not change any name within the given `IASTTemplateId` instance. It just copies all nodes within the template name and returns a copy of it. The copy can then be used in a rewrite change.

InlineTemplateIdFactory This factory adds qualifiers to names within the given `IASTTemplateId` instance if required. The names that require to be changed are provided by a context object.

ExtractTemplateIdFactory This factory removes qualifiers that are not required any more, for instance, because they will be placed in a using declaration or using directive. The names that require to be changed are provided by a context object.

3.3.5. Name Lookup Algorithm

One fundamental question that applies for both, the inline and extract refactorings, is how the look up of the `IASTName` nodes, that require to be changed, is implemented. After some discussions we worked out two different approaches, each with its pros and cons. The analysis and approaches are discussed in the following sections.

Figure 3.6 illustrates the different scopes that play a role when comparing the two ap-

proaches. The figure uses sample names, that would play a role with an inline refactoring, to illustrate the meanings of the scopes. Each scope represents a set of IASTName nodes. The sets are:

Source Names that must be considered when searching for candidates. E.g. the name of the function definition `x` and the namespace definition `A`.

References Names that reference one of the names in the source set. These are references found by the indexer, and therefore can be anywhere in the active workspace (even in system libraries). Any found `A` or `x`;

Scope Names in the scope that the selection is in. Any found `A` or `x`;

Target Names within the scope set that require to be changed, e.g. qualified or extracted. For an inline refactoring this would be the name `x`.

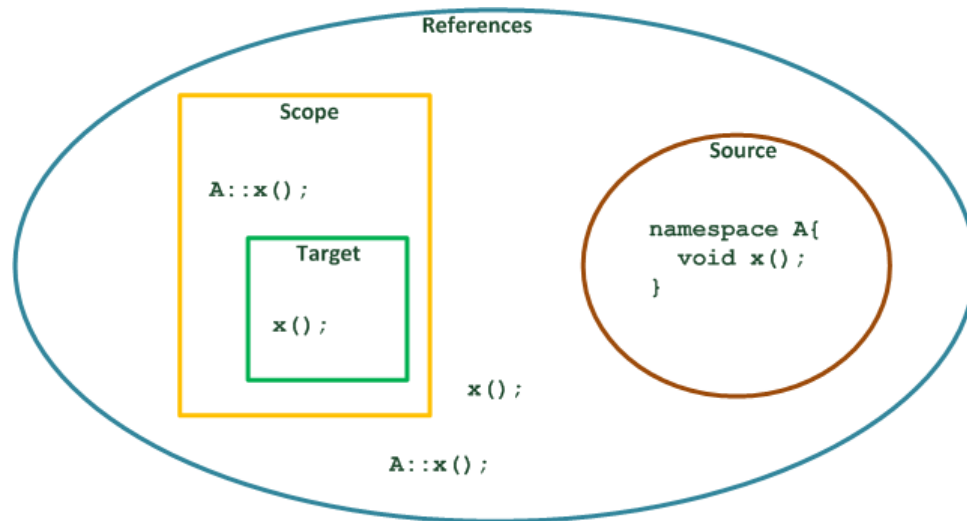


Figure 3.6.: Scopes that play a role when comparing the indexer and AST lookup algorithms.

The aim of the lookup algorithms, described in the sections below, is to create the target set. All names therein will then be processed, e.g. qualified or extracted.

3.3.5.1. Indexer Lookup

The indexer lookup algorithm uses the source set as input. Given a set of names it uses the indexer to search all references of all source names. Then it continuously filters the reference set based on different criteria until only the target set is left. First, it filters out names that are within the same scope as the selection is in. Then, a validator mechanism is used to check if a given name requires to be changed (e.g. qualified or extracted). If so it is added to the target set.

3.3.5.2. AST Lookup

The AST lookup algorithm uses the scope set as input. It uses a visitor [GHJV95] to collect all names within the scope of the selection. The binding owner of each visited name is compared to the binding of the namespace to be extracted (EUDIR) or to the potential type to be extracted (EUDEC) within the source set. A validator mechanism is used to check if a given candidate name requires to be changed (e.g. extracted with EUDIR or EUDEC).

3.3.5.3. Implementation Strategy

We decided to start our implementations using two strategies. The inline refactorings started with implementing the indexer lookup, the AST lookup was implemented by the extract refactorings. The reasons, why we decided to implement both algorithms in parallel, are as follows:

- The knowledge of the CDT API was not sufficient to decide whether or not one approach will definitely work or not.
- During the semester (and this) thesis we have experienced that CDT has still fundamental bugs (e.g. 3.4.5, 3.4.2).
- By implementing two alternative lookup algorithms in parallel, we minimize the risk of having to start over if one approach fails. If so, the knowledge gained during the parallel development of the other implementation could be used to quickly adapt the other refactoring to the working algorithm.

3.3.5.4. Result

The implementation of the indexer based lookup algorithm in the inline refactorings, described in section 3.3.6, revealed the following issues:

Complexity Filtering the reference set to gain the scope set is a waste of time. The AST approach does not require this step. The percentage of filtered names is potentially very low, since the indexer scope can be much bigger than, for example, the scope of a function.

Scalability Using the indexer to get all references can result in a fairly big amount of names, especially if working with system or third party libraries. The indexer based algorithm creates an `IASTTranslationUnit` for every file a referencing name was found in. Creating an AST, possibly hundreds of times, requires much time and memory and should therefore be avoided.

AST of std Namespace Even if we agree with creating an AST for each system include, getting the AST node of the std namespace definition is tricky, because it is created using macros. namespactor does not plan to support macros. Since the std namespace will likely be inlined very often, the indexer approach does not fit here.

Based on the issues listed above, after the planned code freeze, we decided to abandon the indexer lookup and try to adapt the IUDIR refactoring to the AST lookup algorithm. We managed to implement the new lookup algorithm for the IUDIR refactoring in two days. But we could not use a plain AST lookup, instead we had to create a new approach that uses the indexer and the AST. This new hybrid approach is described in paragraph 3.3.5.4.1.

Because we were running out of time we did not change the indexer based implementations of the IUDEC refactoring implementation, which is described in section 3.3.6.5. A detailed description of the new hybrid implementation for the IUDIR refactoring is given in section 3.3.6.4. The description for the now obsolete IUDIR refactoring implementation using the indexer can be found in appendix D.

3.3.5.4.1. Hybrid Lookup The hybrid lookup algorithm combines the pros of the indexer lookup as well as the pros of the AST Lookup. It uses the scope set as input. A visitor is used to collect all names within the scope of the selection. The index binding of each visited name is compared to all bindings of the namespace definition index names within the source set. Equal names are collected as candidates for the target set. A validator mechanism is used to check if a given candidate name requires to be changed (e.g. qualified with the IUDIR refactoring). If so, it is added to the target set.

3.3.5.4.2. Why Hybrid? The IUDIR refactoring requires access to nested using directives within namespace definitions to support transitive using dependencies, see section 3.1.3.4.4 for more details. Nested using directives can be accessed by using the `ICPPASTNamespace.getDeclarations()` method. But to get an `ICPPASTNamespace` instance, an AST is required. As already mentioned above, getting the AST of every file (also system includes) is very inefficient.

Another way to access using directives within namespace definitions is by using the method `ICPPNamespace.getUsingDirectives`. `ICPPNamespace` instances can be retrieved from the indexer. Unfortunately the method `getUsingDirectives` is not implemented as expected, as described in the open issues section 3.4.2.

The hybrid lookup uses the indexer to get all namespace definition index names of the namespace nominated by the selected using directive. The AST is used to visit the scope of the selected using directive.

3.3.6. Inline Refactorings

This section reveals the implementation details of the inline refactorings inline using directive, inline using declaration and qualify an unqualified name. These implementations are based on the definitions in section 3.2. First, each refactoring is illustrated based on a simple C++ code snippet 3.46. After that, each refactoring implementation is explained in detail in separate sub sections.


```
namespace Collections{
    struct Element{
        Element(int value){}
    };
}
```

Listing 3.46: Foundation for the refactoring demonstration snippets 3.47, 3.48 and 3.49.

3.3.6.1. Inline Using Directive

Qualifies all affected names in the scope of the selected using directive with its name and removes it.

```
// before
using namespace Collections; // apply inline refactoring here
Element e1(23);

// after
Collections::Element e1(23);
```

Listing 3.47: Before and after illustration of the inline using directive refactoring based on a simple example snippet (3.46).

3.3.6.2. Inline Using Declaration

Qualifies all affected names in the scope of the selected using declaration with the name of the using declaration and removes it.

```
// before
using Collections::Element; // apply inline refactoring here
Element e2(42);

// after
Collections::Element e2(42);
```

Listing 3.48: Before and after illustration of the inline using declaration refactoring based on a simple example snippet (3.46).

3.3.6.3. Qualify an Unqualified Name

Qualifies the selected unqualified name. Only works with unqualified names. A future improvement is to allow the full qualification of an already qualified name.

```
// before
Element e2(42); // apply inline refactoring here

// after
Collections::Element e2(42);
```

Listing 3.49: Before and after illustration of the qualify an unqualified name refactoring based on a simple example snippet (3.46).

The following sections focus on the details of the three inline refactoring implementations. In section 3.3.6.7 different aspects that are common to multiple inline refactoring implementations are explained.

3.3.6.4. Inline Using Directive Refactoring - Hybrid Implementation

The indexer based implementation of the IUDIR refactoring was started after we discovered that the indexer based implementation, described in appendix D, had some disadvantages. See section 3.3.5 for an overview about the pros and cons of both approaches.

This implementation has some open issues related to template method definitions and implicit operator calls. These are described in detail in the sections 3.4.4 and 3.4.3 respectively.

The pseudo code in listing 3.50 summarises the algorithm implemented by the IUDIR refactoring.

```
selectedUsing = getSelectedUsing(selection);
usingScope    = getScopeOf(selectedUsing);
namespacesPerUsing = findNamespaceDefinitionsRecursively(selectedUsing);
targetsPerNamespace = findTargetsInScope(usingScope);
validTargetsPerNamespace = filterValid(targetsPerNamespace);
inlineDeclarationName(validTargetsPerNamespace);
removeUsing(selectedUsing);
```

Listing 3.50: Pseudo code illustrating the algorithm used in the hybrid based IUDIR refactoring implementation.

Figure 3.7 shows a sequence diagram that gives an insight on how the IUDIR implementation looks like. The methods are explained in the following:

checkInitialConditions This method is invoked by the LTK framework prior to show the refactoring wizard. If the returned refactoring status contains an error the wizard is not shown, instead the error is displayed to the user. See section 3.3.2 for more details.

getSelectedUsing Gets the selected using directive. It uses a visitor to visit all declarations within the active IASTTranslationUnit and returns the last one that is within the offsets of the active text selection provided by Eclipse. If null is returned, no using directive is selected and the user is notified appropriately.

findCompoundStatementInAncestors Recursively traverses the AST upwards, starting at the selected using directive, until an IASTCompoundStatement is found, if not, null is returned. The compound statement is required to validate if a candidate reference node is in the same scope as the originating using directive where the refactoring was initiated from. See method findTargetsInScope. If no compound is returned, the IASTTranslationUnit is used as scope.

findNamespaceDefinitionsRecursively Uses the indexer to get all definitions of the namespace name nominated by the using directive. All of them are added to a map (namespacesPerUsing) that holds a list of namespace definition names per namespace name. If a namespace definition contains another using directive, the namespace nominated by it would be searched as well. Unfortunately there is a bug in CDT that does not allow to retrieve using directives nested within a namespace definition, see 3.4.2.

findTargetsInScope Uses an ASTVisitor to visit all name nodes within the scope the selected using directive is in. The aim is to create a set of target names that require to be changed. If the using directive is in the global scope all include dependent translation units are visited as well. More about include dependencies can be found in section 3.3.6.7.1.

A name is ignored if it matches any of the following criteria:

- The name is ignored if it is enclosed by one of the source namespace definitions. Visiting such a name could happen if the using directive is in the global scope. In such a case the namespace definition, nominated by the using directive, is visited as well, but all names within that namespace definition can be ignored, since they do not require to be qualified. As soon as a namespace definition is visited its IASTName is compared to all collected IIndexName namespace names in the namespacesPerUsing map. If one matches, the namespace definition is skipped. To compare whether an IIndexName represents the same name as an IASTName their positions are compared. Their node offset, node length and file name must be equal.
- If the binding owner of the visited name is null, the name is ignored, because than it is a name from the global scope.

A name is elected as target name if it matches all of the following criteria:

- If an enclosing compound is given, the name must be within the enclosing compound

- The indexer binding owner of the name is equal to one of the indexer bindings of the namespace definition names from the source.
- The name is not yet qualified

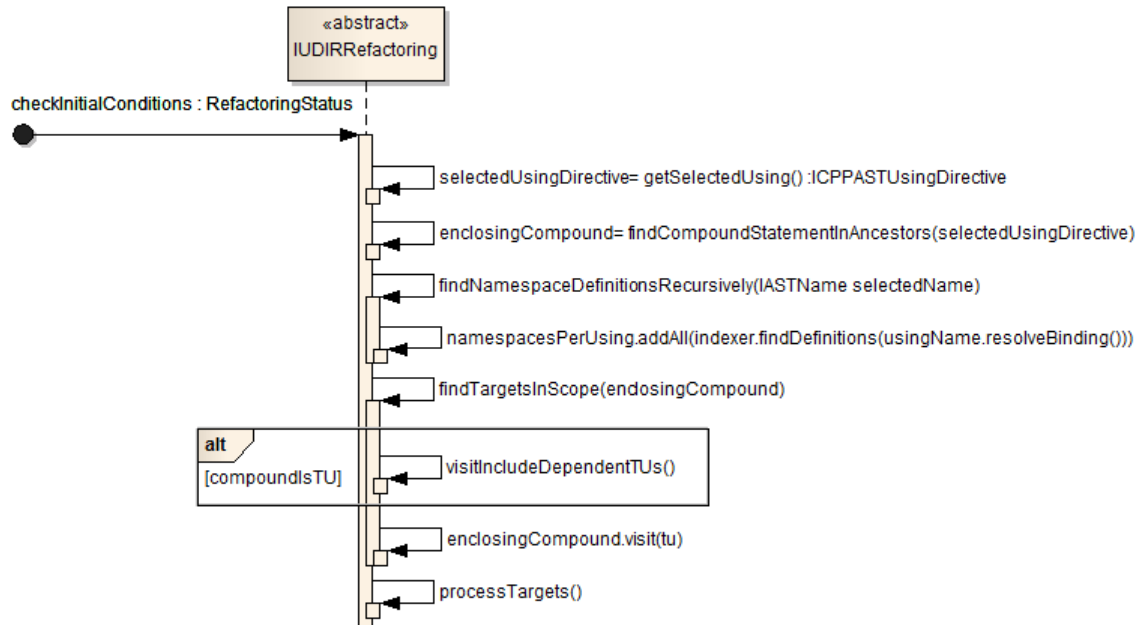


Figure 3.7.: Sequence diagram of the IUDIR refactoring AST implementation.

3.3.6.5. Inline Using Declaration Refactoring Implementation

The IUDEC implementation follows the same basic indexer based algorithm, as the first approach of IUDIR refactoring implementation uses, which is explained in appendix D. The main difference is that with inline using declarations only the references of one declaration, the one introduced with the using declaration, must be considered to be qualified. In contrast to using directives where candidates have to be looked up recursively. See appendix D.1 for more details on recursive search.

Using declarations require special treatments if templates are in place, as explained in the following paragraphs 3.3.6.5.1 and 3.3.6.5.2. The IUDEC refactoring currently does not work for names defined outside the workspace, see 3.4.1.

3.3.6.5.1. References of Template Names Template names in using declarations require special treatment. For example when used inside composite types with multiple inheritance as shown in listing 3.51. See paragraph 3.1.3.4.5 for more details on multiple inheritance.

The default approach to lookup references of using declaration names is to simply call `Index.getReferences(IBinding)` where `IBinding` is the binding of the name of the selected using declaration. But with template names this does not work, an empty array

is returned.

To find the declaration, the name of a using declaration points to, the method `ICPPUsingDeclaration.getDelegates()` is used. `ICPPUsingDeclaration` is the casted `IBinding` of the using declaration name binding. This method returns an array of `IBinding`, where each of it is a reference to a previously declared `IBinding`. In this case, the first delegate represents the method definition `S::a()`. Using the delegate binding, the method `Index.getReferences(IBinding)` is able to find the call site within method `T::b()`.

The code in listing 3.51 will not compile with GCC versions less than 4.5.2, because the using declaration uses a concrete template argument (`char`). This is documented as open issue in section 3.4.7. If a user tries to inline a using declaration with a generic template argument, the refactoring wizard shows a warning explaining the issue (see figure 3.8).

```
namespace AA{
    template <typename T>
    struct S{
        void a(){} // only found if delegates are used
        template<class F> void t(){}
    };
    struct U{
        void a(){}
        template<class F> void t(){}
    };
    template <typename C>
    struct T : public S<char>, public U{
        using S<char>::a; // start inline refactoring here
        using U::t;
        void b(){
            a(); // qualified with S<char>
            t<int>();
        }
    };
}
```

Listing 3.51: Using declaration with template name in a multiple inheritance scenario.

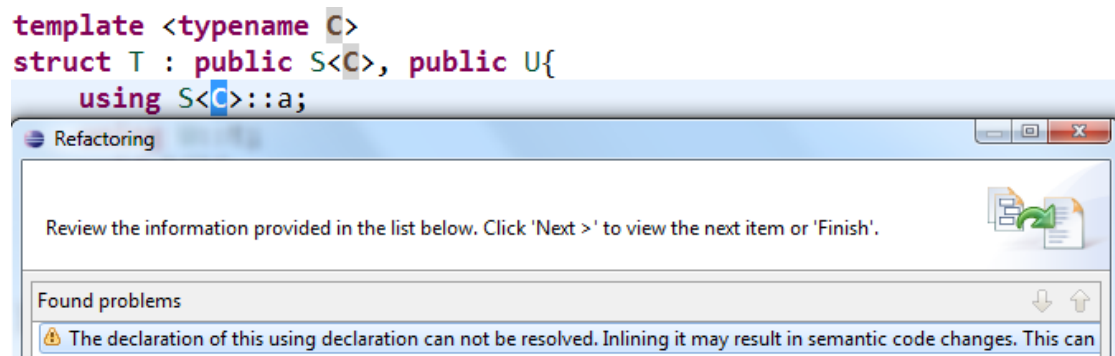


Figure 3.8.: Refactoring wizard shows a warning if the user tries to inline a using declaration that contains a generic template argument.

3.3.6.5.2. References of Template Function Calls If a using declaration declares a template function, special treatment is required to find all referencing function calls. First, the delegate mechanism described in paragraph 3.3.6.5.1 is used to find the declaration the using declaration name points to. Then, the references of the found function definition are looked up.

The default approach is to simply use `IIndex.findReferences(IBinding)`, where `IBinding` is the binding of the function declaration. But this approach does not always work for template functions, as illustrated in listing 3.52. The first function call `getMax(1, 2)` is not found as reference, but the explicit function call `getMax<int>(1, 2)` is found.

In C++, a template function has multiple instances. One for each specialisation. In listing 3.52 only one specialisation exists, that is for the template argument `int`. The solution is to look for references of `ICPPTemplateInstance` bindings. To get all instances of a template function definition, the method `ICPPInstanceCache.getAllInstances` is used, where `ICPPInstanceCache` is the casted `IBinding` of the function definition. Calling `IIndex.findReferences(ICPPTemplateInstance)` also finds call sites without explicit template arguments, in this case `getMax(1, 2)`.

```
namespace AA{
    template <class T>
    T getMax(T a, T b){return a;}
}

template <class T>
T getMax(T a, T b){return a;}

int main(){
    using namespace AA;
    using AA::getMax; // start inline refactoring here
    getMax(1, 2); // only found when using ICPPTemplateInstance
    getMax<int>(1, 2); // always found as reference and qualified with AA
    return 0;
}
```

Listing 3.52: Using declaration pointing to a template function.

3.3.6.6. Qualify an Unqualified Name Refactoring Implementation

The QUN refactoring implementation is the most simple of the inline refactoring implementations. This is because the name to be inlined is already given as input to the refactoring. Therefore, no references require to be found, which is a challenging part in the IUDIR and IUDEC refactoring implementations (see appendix D and section 3.3.6.5 respectively).

But since the QUN refactoring also changes names in the AST, it has to be able to modify template names. How this is done is described in section 3.3.4. The QUN refactoring is currently not able to modify template names defined outside of the workspace, see 3.4.1 for more details.

3.3.6.7. Common Aspects

This section outlines aspects that are common to multiple refactoring implementations.

3.3.6.7.1. IUDIR, IUDEC - Indexer Scope The IUDIR (see appendix D) and IUDEC (see section 3.3.6.5) refactoring implementations use the indexer to search for references and definitions of IBindings.

For example, all definitions of a namespace definition binding of a namespace that was nominated by a using directive. This kind of lookup is done by the recursive search algorithm of the IUDIR refactoring, see appendix D.1.

To find definitions of an IBinding, the indexer compares the IIndexBindings of the found nodes. This bindings equal if they have the same name and are of the same type. The CRefactoring2 base class of the refactorings provides an indexer that indexes all open projects within the active workspace. Considering that two projects exist, each with a definition of namespace A, and no dependencies exist between the projects, the indexer will return both definitions. This is not desirable for the inline refactorings, since nodes in foreign projects may be affected, although they are not within the same translation unit.

The same effect also applies for two files within the same project. Although they have no dependencies to each other, the indexer will search both of them and only look for equal IIndexBindings. Listing 3.53 illustrates the results of an indexer search.

```
// file A.cpp
// searching definitions of this namespace definition with the indexer
// returns: A.cpp::A, C.h::A and B.cpp::A
// desired: A.cpp::A, C.h::A
#include "C.h"
namespace A{ }

// file B.cpp
namespace A{ }

// file C.h
namespace A{ }
```

Listing 3.53: Indexer does not respect #include dependencies when searching for definitions.

To solve that problem a dependency analyser is used, which is implemented in the class IncludeDependencyAnalyser that provides the following methods:

areFilesIncludeDependent(IIndexFile file, String originFileName) Returns true if *file* includes *originFileName* or if *file* is included by *originFileName* in any depth.

This method is used to check whether two nodes are in the same file or within files that include each other. If true, the files belong to the same translation unit and must be further processed. If not, the found definition can safely be ignored, because, for the compiler they do not belong to each other. The IIndexFile instance is retrieved by calling the IIndexName.getFile() method. An array of IIndexName

is returned from the method `IIndex.getDefinitions(IBinding)`, where `IBinding` is the binding of a namespace definition for instance.

This method is used by the IUDIR (see appendix D) and EUDEC (see section 3.3.6.5) indexer based implementations.

getIncludeDependentPathsOf(ITranslationUnit tu) Returns a List of `IPath`. A file (path) is added to the list if the file is included by the `tu` or the `tu` is included by the file. Only paths that point to a file within the workspace are added to the list. This method is used by the IUDIR hybrid implementation, see section 3.3.6.4.

By using the `IncludeDependencyAnalyser` class, it is possible to profit from the power of the indexer to search the code base and at the same time reduce the scope of the target set to nodes that belong to the same translation unit.

3.3.6.7.2. Indexer File Set An indexer instance contains a set of files. An indexer instance can be created to index any number of open projects within the workspace. Most likely an indexer over all projects is used. Any implementation file within the indexed projects is added to the indexer. But only header files that are included by another file (either header or implementation) are added to the indexer.

3.3.7. Extract Refactorings

This section reveals the implementation details of the extract refactorings `extract using directive` and `extract using declaration`. The implementations are based on the definitions in section 3.2. First, each refactoring is illustrated based on the simple C++ code snippet 3.54. After that, the interesting algorithms of the implementation are explained in detail.

```
namespace Collections{
    struct Element{
        Element(int value){}
    };
}
```

Listing 3.54: Foundation for the refactoring demonstration snippets 3.55 and 3.56.

3.3.7.1. Extract Using Directive

Introduces a `using` directive for the selected qualified name and removes the name qualifier(s) from the affected qualified names.


```
// before
Collections::Element e1(23); // apply extract refactoring here

// after
using namespace Collections;
Element e1(23);
```

Listing 3.55: Before and after illustration of the extract using directive refactoring based on the simple example snippet 3.54.

3.3.7.2. Extract Using Declaration

Introduces a using declaration for the selected qualified name and removes the name qualifiers(s) from the occurrences of the affected qualified name.

```
// before
Collections::Element e2(42); // apply extract refactoring here

// after
using Collections::Element;
Element e2(42);
```

Listing 3.56: Before and after illustration of the extract using declaration refactoring based on the simple example snippet 3.54.

3.3.7.3. Extract Refactorings - Base Algorithm

This section focuses on the details of the two extract refactoring implementations. The EUDIR and EUDEC refactorings both follow the same basic algorithm, which is shown as pseudo code in listing 3.57. The challenges of the EUDIR and EUDEC refactorings are explained separately in the following sections.

```
// only qualified names are interesting
selectedQualifiedName = getSelectedQualifiedName(selection);
// name for using directive/declaration
usingName = buildUsingNameFrom(selectedQualifiedName);
scopeOfInterest = findScopeOf(selectedQualifiedName);
visitNodesIn(scopeOfInterest){
    if(node is a replace candidate){
        replaceNode = buildReplacementNodeFor(node);
        // using directives could be useless cause of new one
        removeUsingIfUseless(node);
        replaces = addReplace(replaceNode);
    }
}
// correct placement of the using directive/declaration
insertionPoint = findFirstNodeToReplace();
insertUsingStatement(scopeOfInterest, insertionPoint, usingName);
applyReplaceChanges();
```

Listing 3.57: Pseudo code for the EUDIR and EUDEC algorithms.

The following sections describe interesting parts of the extract refactorings. The sequence diagrams in the following sections represent abstractions that are valid for both refactorings. Both extract refactorings, EUDIR and EUDEC, follow the same base algorithm explained in the sections below. The two refactorings differentiate only in a few method implementations which are explained in the method descriptions.

The sequence diagrams in each section are not exact representations of the source code, only important and interesting parts and methods will be considered and explained. This also applies to the descriptions of the method implementations.

3.3.7.4. checkInitialConditions - EURefactoring

The method is called by the LTK of Eclipse and initiates the particular extract refactoring, see 3.3.2 for more information about the LTK. Figure 3.9 illustrates the checkInitialConditions implementation. The LTK first invokes this method on the refactoring object to determine whether the refactoring is applicable at all in the context desired by the user [ltk06].

getSelectedQualifiedName Gets the selected qualified name. It uses a visitor to visit all declarations within the active IASTTranslationUnit and returns the one that is within the offsets of the active text selection. If null is returned, the selection is invalid.

This method basically uses a selection helper class provided by Eclipse. To determine the exact selection inside a template and its potentially nested qualified names, our method `NSSelectionHelper.getInnerMostSelectedNameInExpression` is used, which is an extended version of `SelectionHelper.isSelectionOnExpression`.

Consider a qualified name `A::B::S1<A::B::S2>` with a selection on `A::B::S2`. `SelectionHelper.isSelectionOnExpression` (Eclipse) returns true because `A::B::S2` is inside of `A::B::S1<A::B::S2>`. Our extended implementation returns the qualified name `A::B::S2`. This is important since all further operations depend on the

selected qualified name found here.

buildUsingNameFrom Builds the name for the new using directive/declaration. The name is built with all qualifiers even if the selected qualified name is not qualified with all qualifiers. Consider a function `f` inside a namespace `B` that is nested in a namespace `A`. If the qualified name of a function call `f()` is `B::f`, the name of the extracted using directive will be `A::B`. The same applies for the extract using declaration that will result in the name `A::B::f`. The following code snippet 3.58 illustrates this behaviour.

```
namespace A{
    namespace B{
        void f();
    }
}
// before extract refactoring
void func(){
    using namespace A;
    if(true){
        B::f(); // apply refactoring here
    }
}
// after extract refactoring
void func(){
    using namespace A;
    if(true) {
        using namespace A::B; // new introduced using directive
        f();
    }
}
```

Listing 3.58: Extract a using directive with all qualifiers of the selected qualified name.

This method is implemented different for each extract refactoring. The EUDIR refactoring extracts only namespace names into a using directive. The EUDEC refactoring on the other hand allows the extract other names like function names or type names. It is also possible to extract template names to using declarations. EUDEC also implements a special treatment if the extract using declaration is inserted into the scope of a type declaration (ICPPASTCompositeTypeSpecifier). This case is further discussed in the analysis section 3.1.4.3.

findScope Finds the scope of the selected qualified name. This method calls sequentially the methods `findCompoundScope`, `findTypeScope`, `findNamespaceScope`. All three methods search in the ancestor nodes of the `ICPPASTQualifiedName`. `findCompoundScope` returns an `IASTCompoundStatement` if existing. `findTypeScope` returns an `ICPPASTCompositeTypeSpecifier` if existing. `findNamespaceScope` returns an `ICPPASTNamespaceDefinition` if existing. If all three methods return null, the `IASTTranslationUnit` node is the scope. `findTypeScope` is only implemented for the EUDEC refactoring, since it is not possible to place a using directive inside a type declaration.

acceptReplaceVisitor Starts the visitation on the node detected with the `findScope`

method.

getReplaceVisitor Depending on which refactoring was initiated, this method returns an implementation of the abstract EUReplaceVisitor.

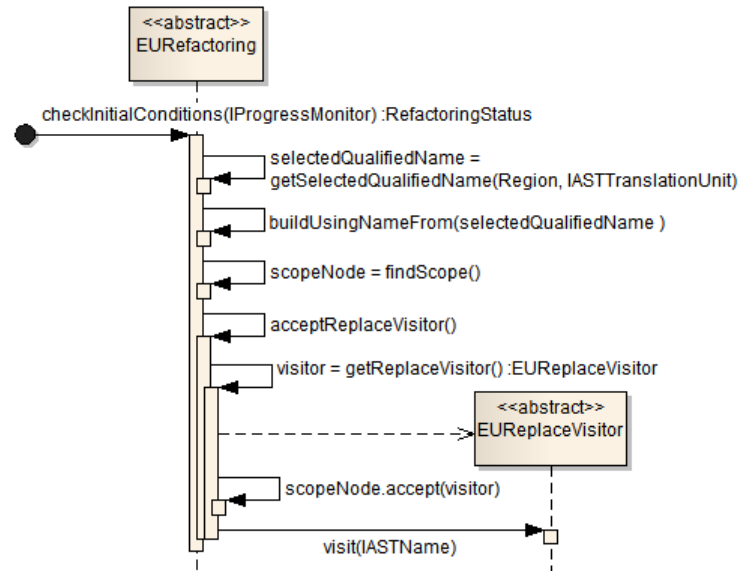


Figure 3.9.: Sequence diagram illustrating the `checkInitialConditions` implementation of the extract refactorings.

3.3.7.5. visit - EUReplaceVisitor

This method is called by `checkInitialConditions` and visits all names in the scope of interest. Figure 3.10 illustrates the visit implementation.

buildReplacementName Builds the replacement name for each relevant name. This method is explained in detail in figure 3.11 and its method description 3.3.7.6.

removeUselessUsingDirective Removes a useless using directive, if, because of the replacement, the using directive has no more value. If an extracted using directive introduces the same namespace as a using directive in the same scope already does, this redundant using directive is removed.

replace Prepares the replacement of the origin name with the replace name in the `ASTRewriteStore` 3.3.2.

removeUnqualifiedUsingDirective Especially handles using directives with unqualified names, since in all other cases only qualified names are relevant for a replacement. This method calls the methods `buildReplacementName` and the `removeUselessUsingDirective` explained above. This method is only implemented for the `EUDIR` refactoring.

Additionally the visit method prepares the insertion point for the extracted using directive/declaration. The first name to be changed in the scope of interest is the input for the insertion point search algorithm. The definitive insertion point is determined in the collectModification implementation, see section 3.3.7.7.

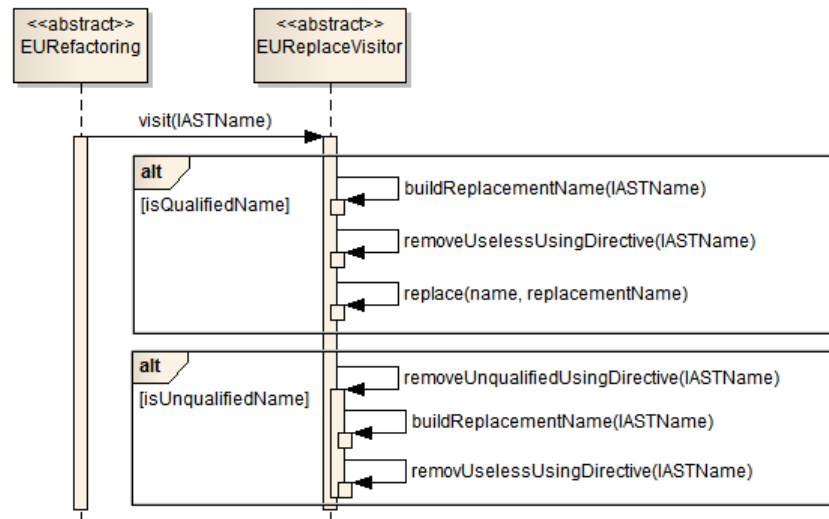


Figure 3.10.: Sequence diagram illustrating the visit implementation of the extract refactorings in the abstract EUREplaceVisitor class.

3.3.7.6. buildReplacementName - EUREplaceVisitor

This method is called by the visit method, described above, and is responsible for the correct name building of the name of interest. Figure 3.11 illustrates the buildReplacementName implementation.

alt isExtractCandidate Determines if the name is a candidate that is affected by the extract refactoring. For example, if the name is a child of a function call expression or a parameter declaration. An invalid example would be a function declaration, because a function declaration can not be declared as qualified name.

getNamesOf Gets the names of the name to be replaced. Especially relevant if the name is a qualified name.

alt isReplaceCandidate The default implementation identifies if a name has a namespace qualifier of interest and therefore is a candidate to be replaced. This method is implemented different for each extract refactoring. EUDEC additionally requires to check if the last name is the same name as the one extracted into a using declaration, this is achieved by EUDEC overriding a default null implementation of another method only used by isReplaceCandidate.

loop names Loop over all names returned by getNamesOf.

alt isTemplateReplaceCandidate Determines if the current name is a template name

and therefore maybe a template name of interest to be replaced.

buildReplacementTemplate Builds the replacement name for a template name. Further information about template names are in the Building Names section 3.3.4. This method is implemented different for each extract refactoring.

replaceName.addName If the namespace of interest is found in the names loop, the replacement starts and the following names in to loop are added to replaceName. This replaceName is returned at the end of the buildReplacementName method.

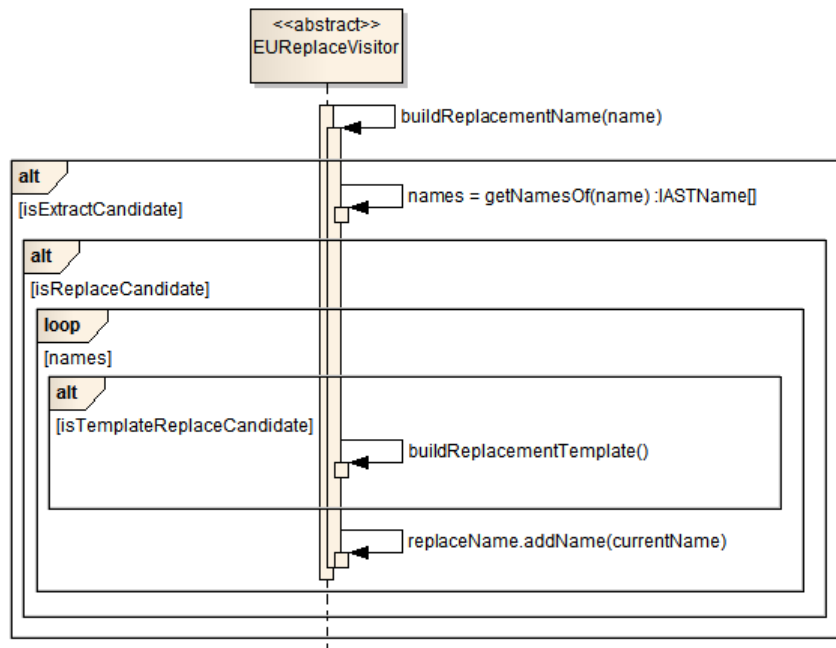


Figure 3.11.: Sequence diagram illustrating the buildReplacementName implementation of the extract refactorings in the abstract EUNamespaceVisitor class.

3.3.7.7. collectModifications - EUNamespaceRefactoring

This method is called by the LTK of Eclipse and collects all the changes to be performed in the AST, see 3.3.2 for more information about the LTK. Figure 3.12 illustrates the collectModifications implementation.

addReplaceChanges Adds all replace changes, collected in a context object, during the visits in the scope of interest, see figure 3.10, to the ASTRewriteStore, described in appendix A.7).

addInsertChange First calls the findInsertionPoint and prepareInsertStatement methods explained below then adds the insert change to the ASTRewriteStore.

findInsertionPoint On the basis of the first name to be changed, determined in the visit implementation, illustrated in figure 3.10, the place to insert the new using directive/declaration is found here. The node where the using directive/declaration

has to be inserted before, has to be a child in the scope of interest. Therefore, the parent of the insertion point node has to be the same node as the scope of interest. This means that somewhere in the parent hierarchy of the first name node to be replaced, the insertion point is found.

prepareInsertStatement Creates the using directive/declaration with the name built by buildUsingNameFrom in the checkInitialConditions implementation 3.9. This method is implemented different for each extract refactoring.

addRemoveChanges Adds all remove changes, collected in a context object, during the visits in the scope of interest to the ASTRewriteStore.

performChanges Performs all the changes that the refactoring added to the ASTRewriteStore.



Figure 3.12.: Sequence diagram illustrating the `collectModification` implementation of the extract refactorings in the abstract `EUNRefactoring` class.

3.4. Open Issues

This section describes the issues left open at the end of this bachelor thesis. Some of them are also documented in the issue tracker at [nam12]. If so the section heading features the issue number.

3.4.1. Qualify Names Defined Outside of the Workspace - #273

The IUDEC (section 3.3.6.5) and QUN (section 3.3.6.6) still use the infrastructure that was introduced by the indexer based IUDIR implementation (see appendix D). Therefore, they have difficulties in handling names that are defined outside the workspace, see 3.3.5.4 for more details. For the QUN refactoring this means, that inlining template names defined outside the workspace is not possible, e.g. `vector<string>`. With the IUDEC refactoring it is not possible to change any name that is defined outside the workspace. In both cases the user is notified with an appropriate error message explaining the issue.

The indexer based implementations heavily rely on the method `RefactoringBase.getNodeOf(IName, IProgressMonitor)`, which gets the `IASTNode` of a given `IIndexName`. This method fails to get an `ASTNode` if it was defined in a `IASTTranslationUnit` located outside the workspace.

3.4.2. Nested Using Directives within Namespace Definitions - #269

Using directives nested within namespace definitions can be accessed by using the `ICPPASTNamespace.getDeclarations()` method. Another way, offered by the PDOM implementation, is to use the method `ICPPNamespace.getUsingDirectives`. `ICPPNamespace` instances can be retrieved by using the indexer. Unfortunately the method `getUsingDirectives` is not implemented as expected. Always an empty array of `ICPPUsingDirective` is returned. The bug is reported to CDT [cdt12b].

3.4.3. Finding Implicit Operator Calls - #270

The hybrid based IUDIR implementation does currently not look for implicit operator calls. The indexer based implementation used the indexer to get all references of an operator overload definition. This references also contained implicit operator calls. But the hybrid implementation uses an AST visitor to visit all name nodes within the scope of the selected using directive. Hence, for example the implicit shift operator call (`«`) is never found, because in the AST it is represented as `IASTBinaryExpression`. Even if the visitor would visit such an operator call, it can not simply be treated as name that could be qualified. Instead it has to be transformed into an explicit operator that can then be qualified. The indexer based IUDIR implementation has a similar issue, see appendix D.2.1.

3.4.4. Qualifying Template Method Definitions - #271

The hybrid based IUDIR implementation currently does not support to qualify unqualified template method definition names. The indexer based implementation did support this feature. Something went wrong by adopting the hybrid based lookup algorithm. Further research is required to find the exact cause of the failure.

3.4.5. Missing Line Break after last Affected Node - #238

Both extract refactorings, EUDIR and EUDEC, introduce one new node (a using directive or a using declaration) and replace names in the AST. Before the last node that is affected by a replace change, always a line break is missing. Listing 3.59 illustrates this misbehaviour.

```
// before the extract using directive refactoring
void func(){
    A::a(); // apply refactoring here
    A::B::b();
    A::a();
}
// after the extract using directive refactoring
void func() {
    using namespace A;
    a();
    B::b();a(); // missing line break before a();
}
```

Listing 3.59: Missing line break after last affected node.

This misbehaviour is not fatal so far. But if there is a one line comment before the last node to be affected, the resulting code changes its behaviour which is not the idea behind refactorings and it could lead to compile errors as well. Listing 3.60 illustrates this more precarious misbehaviour.

```
// before the extract using directive refactoring
void func(){
    A::a(); // apply refactoring here
    A::B::b();
    // one line comment
    A::a();
}
// after the extract using directive refactoring
void func() {
    using namespace A;
    a();
    B::b();
    // one line comment a(); -> function call a() is now in the comment
}
```

Listing 3.60: Missing line break after last affected node, function call `a()` is now in the comment.

The problem appears with the usage of the `ASTRewriteStore`, see section A.7, as well as with the usage of `ModificationCollector` that is used by the LTK, see section 3.3.2. The `ASTRewriteStore` is a wrapper for the `ModificationCollector`.

It is also possible to perform AST modifications using the `ASTModificationStore` of Eclipse CDT but this issue is not reproducible with the `ASTModificationStore`.

Since we use the LTK in our refactoring implementations and it is not in our competence to change the `ModificationCollector`, this issue remains open.

3.4.6. Creating Fully Qualified Names - #249

The `ICPPQualifiedName.setFullyQualified` method allows to mark a qualified name as fully qualified. The implementations are correct. But the `ASTWriter` does not respect the flag when writing qualified names. Hence, the leading scope resolution operator `::` is missing. This issue is reported to CDT ([cdt12a]).

3.4.7. Using Declaration with Generic Template Argument - #239

The code in listing 3.61 will not compile with GCC versions less than 4.5.2, because the using declaration uses a concrete template argument (`char`). Lower versions of GCC require the template argument to be generic. But with generic template arguments the reference lookup, as described in paragraph 3.3.6.5.2, does not work, because the returned binding is an `ICPPUnkownBinding`. Changing this behaviour is not in our competence. Therefore, inlining using declarations with generic template arguments are not supported for now. If a user tries to inline a using declaration with a generic template argument, the refactoring wizard shows a warning explaining the issue.

```
namespace AA{
    template <typename T>
    struct S{
        void a(){}
    };
    template <typename C>
    struct T : public S<char>{
        // declaration of S::a not found if C is used instead of char.
        using S<char>::a;
    };
}
```

Listing 3.61: Using declaration with template name.

3.4.8. Inherited Type Name cannot be Replaced - #231

Types can extend other types. The type name(s) of the base type can be qualified with namespace qualifiers and type qualifiers. Listing 3.62 illustrates an example of that issue.

```
namespace A{
    struct S1{};
}
struct S2 : A::S1{ // apply EUDIR refactoring here (A::S1)
    void f(A::S1);
};
```

Listing 3.62: Foundation snippet of the issue: Inherited Type Name cannot be Replaced.
A::S1 is a qualified name of the type S1 which is the base type of S2.

The EUDIR refactoring results in the source code illustrated in figure 3.63.

```
namespace A{
    struct S1{};
}
using namespace A;
struct S2 : A::S1{ // is still qualified with A
    void f(S1); // qualifier A is removed
};
```

Listing 3.63: Result snippet of the issue: Inherited Type Name cannot be Replaced.
Foundation illustrated in listing 3.62

The problem is that in the rewrite algorithm of the AST in CDT, the replace changes of the ICPPASTBaseSpecifier get lost. Hence the result is: struct S2 : A::S1, instead of struct S2 : S1.

3.5. Future Improvements

This section describes the future improvements of this bachelor thesis. Some of the issues are also documented in the issue tracker at [nam12]. If so, the section heading features the issue number.

3.5.1. Implement Hybrid Lookup in IUDEC and QUN

The IUDEC and QUN refactoring still rely on the indexer infrastructure introduced by the indexer based IUDIR implementation. Because we ran out of time these two refactoring could not have been adopted to use the hybrid lookup algorithm, see 3.3.5 for more details.

By updating the IUDEC and QUN implementations, the open issue 3.4.1 will be solved.

3.5.2. Detect Name Conflicts

By applying the EUDIR refactoring it is possible that the refactored code is semantically changed, or that compilation fails. See section 3.1.4.1 and paragraphs 3.1.4.3.5 for more details how name conflicts may be introduced. Currently the EUDIR refactoring is applied without any warning. Possible strategies to handle name conflicts are listed in paragraph 3.1.4.3.6.

3.5.3. Start IUDIR on Any Name

Currently the IUDIR refactoring can only be initiated from a selected using directive. One approach discussed in the analysis is to allow to start the IUDIR refactoring by selecting any name. The Analysis section 3.1.3.4 contains various case studies discussing this approach.

3.5.4. Extract Using Declaration Into a Type Declaration - #265

Using declarations can be used to avoid ambiguous name in case of multiple inheritance, see paragraph 3.1.3.4.5 for more information. The extract refactoring algorithm described in section 3.3.7, does not allow to insert a using declarations into other scopes, than the scope the refactoring was initiated from, which is necessary in this case. An example illustrates the situation in listing 3.64 and 3.65, the desired result is illustrated in listing 3.66.

```
struct U {
    int f(int i);
    char f(char c);
};
struct A : public U {
    int f(int i){}
    char f(char c); // shadow U::f(char)
};
struct B {
    double f(double d) {}
};
```

Listing 3.64: Foundation snippet of the issue: Extract Using Declaration into a Type Declaration from another Scope. The refactoring foundation is illustrated in 3.65.

```
struct AB : public A, public B {
    using B::f; // make B::f(double) accessible from outside
    char f(char c){} // hide A::f(char) (only if 'using A::f' in place)
};
int main(){
    AB ab;
    // A::f(int) exact match, without 'using A::f' => ambiguous, because
    // multiple implicit conversions exist (B::f(double), AB::f(char))
    ab.A::f(1); // apply refactoring here
    // AB::f(char), declarations are: AB::f(char), A::f(char) => no change
    ab.f('a');
    // B::f(double) exact match, without 'using B::f' => ambiguous, because
    // multiple implicit conversions exist (A::f(int), A::f(char), AB::f(char))
    ab.f(3.14);
}
```

Listing 3.65: Foundation snippet of the issue: Extract Using Declaration into a Type Declaration from another Scope.

If the EUDEC refactoring is applied on the call `ab.A::f(1)`, the refactoring should insert a using declaration inside the type declaration of struct `AB` and remove the namespace qualifier `A` from the function call. Listing 3.66 illustrates the desired result.

```
struct AB : public A, public B {  
    using A::f; // new inserted using declaration  
    using B::f;  
    char f(char c){}  
};  
int main(){  
    AB ab;  
    ab.f(1); // namespace qualifier A is removed  
    ab.f('a');  
    ab.f(3.14);  
}
```

Listing 3.66: Result snippet of the issue: Extract Using Declaration into a Type Declaration from another Scope.

3.5.5. Extension for the Refactoring Qualify an Unqualified Name - #265

The implemented QUN refactoring, described in section 3.3.6.6, only works with unqualified names. The same functionality would be feasible and useful with qualified names. Qualified names do not have to be qualified with all qualifiers as illustrated in listing 3.67. The name of the function call `B::f()` is qualified but not qualified with all its qualifiers. Since QUN only works with unqualified names, the name `B::f` is not a candidate for the QUN refactoring as it is implemented now.

If the QUN refactoring would get this additional functionality, it would be renamed to Qualify any Name (QAN).

```
namespace A{  
    namespace B{  
        void f(){}  
    }  
}  
using namespace A;  
int main(){  
    B::f(); // apply QAN refactoring here  
}
```

Listing 3.67: Foundation for the snippet 3.68 related to the issue Extension for the Refactoring Qualify and Unqualified Name.

Listing 3.68 shows the desired result of the QAN refactoring initiated in the code snippet 3.67.

```
int main(){  
    A::B::f(); // is now qualified with A  
}
```

Listing 3.68: Result source code of the refactoring based on the foundation snippet 3.67. The using directive with the name A is removed and the function call is qualified with the name A.

3.6. Conclusion

All objectives specified at the start of this bachelor thesis were achieved. The Codan part described at 3.3.3 was not part of the objectives. Using Codan, we were able to add great value to namespactor, because problem reports explicitly ask the programmer to use one of the namespactor features to enhance the quality of the source code. One of the problem resolutions corresponds to the optional objective "Force Qualified Name in a Using Directive".

We have benefited from the idea to implement the refactorings with two different approaches, outlined in section 3.3.5. Even though some issues are left open, see section 3.4, we gained important knowledge which, at last, led to the hybrid approach. Due to the late code changes we exceeded our deadline of the code freeze, but now we are glad we recognised the misleading approach and implemented the hybrid approach 3.3.5.4.1. Since namespactor was developed only within eleven weeks, it could still profit from further improvements. It would also help to get some feedback from experienced C++ developers to ensure that namespactor can prove itself in real world projects.

4. Metriculator

This chapter contains information that relates to the first phase of this thesis. This is the further development of the metriculator plug-in that was initiated in the semester thesis [met11a]. The first section defines the requirements. Based on the requirements, the design is changed as described in section 4.5. Section 4.2 gives detailed information about performance measurements and improvements made during this thesis. Further sections describe the implementations of the requirements and optional features. At the end, we describe details about our unit test implementations in section 4.7.

4.1. Requirements

Based on the objectives defined in chapter 2 we defined the following requirements.

4.1.1. Performance

4.1.1.1. Current State

At the end of the semester thesis metriculator run out of memory when analysing more than about 350'000 PSLOC. See section 5.1 in the metriculator documentation [met11a] for more details.

4.1.1.2. Objective

metriculator is able to completely analyse 1 million physical source lines of code (PSLOC) in less than 3 minutes without crashing.

The performance tests are always executed in the same environment that has the following specifications:

Computer specifications :

- Processor(CPU): 64Bit, 2.67GHz, Intel Xeon
- RAM: 8GB
- OS: Linux 3.1.9, Fedora release 16 (Verne)

Runtime Eclipse Indigo 3.7.2 with CDT 8.0.2, Java Runtime Environment 6. Depending on the test run, the memory assigned to Eclipse varies.

Metriculator Settings Only the LSLOC metric is activated and problem reporting is

deactivated. Problem reporting is disabled to gain objective results. Whether a problem is being reported or not depends on the workspace properties defined by the user. The LSLOC metric was chosen because it has the most complex implementation.

4.1.2. Tag Cloud - Dealing with Large Data Input

4.1.2.1. Current State

The tag zest cloud component (cloudio [sou11]) produces an error if the input data exceeds an unknown limit. This error is shown as message box to the user. The exact circumstances that lead to the error are not yet determined.

4.1.2.1.1. Objective Independent of the amount of words and their lengths and weights, the tag cloud component creates a tag cloud without any errors.

4.1.3. Composite Update Site

4.1.3.1. Current State

The composite update site was introduced to allow the installation of metriculator, including the tag cloud component, in one single step using the Eclipse update mechanism, without having zest or CDT already installed. At the end of the semester thesis neither the update site nor the composite update did work.

4.1.3.2. Objective

Using a composite update site, users will be able to install metriculator including the tag cloud feature without having any prerequisites installed.

4.2. Performance

In the thesis of metriculator we described some performance issues of the plug-in ([met11a] section 5.1). In this bachelor thesis we aim to further improve the performance. This section reports on the performance improvements made for metriculator.

4.2.1. Performance Measurement - Comparison Before and After the Improvements

Table 4.1 shows a direct comparison of the metriculator performance measurement data before and after the performance improvements, see section 4.2.2 for more details about the improvements.

All the tests were performed using the llvm project [llv11] source code excluding the sub folder at tools/clang/INPUTS. This folder contains test code that forces the llvm parser to produce errors. As a side effect, the CDT parser produces the expected errors as well, hence we ignore that folder.

Before the improvements metRICulator was not able to analyse more than about 400'000 physical source lines of code (PSLOC). Exceeding this limit resulted in a heap out of space error that caused metRICulator to crash. Therefore, the first two rows represent analysis results of the clang sub folder at tools/clang/lib which has 346'377 PSLOC. The remaining three rows represent analysis results of the llvm project source code. The PSLOC of a directory were determined with the command shown in listing 4.1:

```
find . -print | egrep '\.cpp$|\.h$|\.c$' | xargs cat | sed '/^\s*$/d' | wc -l
```

Listing 4.1: Command to get the number of physical source lines of code (PSLOC) ignoring blank lines.

In order to monitor the memory usage of metRICulator, we used the Java tool JConsole [JCo]. To compare and visualise the performance improvements, several screenshots were taken. Each screenshot shows the memory consumption of a code analysis with metRICulator. The analysis of metRICulator started after about one second JConsole started monitoring and ended a few seconds before the screenshot was taken. See the figures 4.1, 4.2 and 4.3 for the screenshots.

PSLOC	VM Memory [MB]	before [s]	after [s]	speedup
346'377	2048	45	12	3.8
346'377	1024	50	12	4.2
1'209'742	2048	-	45	:)
1'209'742	1024	-	64	:)
1'209'742	900	-	120	:)

Table 4.1.: Runtime performance measurement data before and after the improvements. The first column contains the number of the physical source lines of code (PSLOC) that were analysed. There is no before measurement for 1.2 million PSLOC as metRICulator was not able to process that amount of source code.

4.2.1.1. Interpretation

As illustrated in table 4.1 the duration of the analysis depends on the available memory. The more memory is available the faster the analysis completes. The figures 4.1 and 4.2 show the memory consumption over time. Figure 4.3 shows that the memory consumption has improved in the new version compared to the old version. As you can see in the table 4.1 the speed has increased up to four times.

Before the performance improvements it was not possible to analyse more than about

400'000 physical source lines of code (PSLOC). The reason was that there were too many reachable references, that prevented the garbage collector to collect them. Hence, the Java heap space ran out very soon. Before the improvements, the lower limit of memory, to analyse 346'377 PSLOC, was about 1024MB.

After the performance improvements it is possible to analyse more than one million PSLOC without any problem. The lower limit of memory, to analyse 1'209'742 PSLOC, is about 900MB. Running with less memory disproportionately increases the running time. metriculator performs better if there is more memory available, as illustrated in Table 4.1.

4.2.1.2. Further Observations

It is also noteworthy that the metric analysis of metriculator can be executed multiple times without losing its performance. Before the performance improvements metriculator did not release all of its acquired memory after the analysis. Therefore, the process became slower with every further execution.

Activating more than one metric does not significantly decrease the performance. As reasonable, metriculator requires more execution time, but the memory allocation per additionally activated metric is barely apparent.

Running metriculator on llvm with all metrics and problem reporting enabled and 2048MB of memory assigned to Eclipse, takes about 3 minutes to run and allocates at most 1GB of memory.

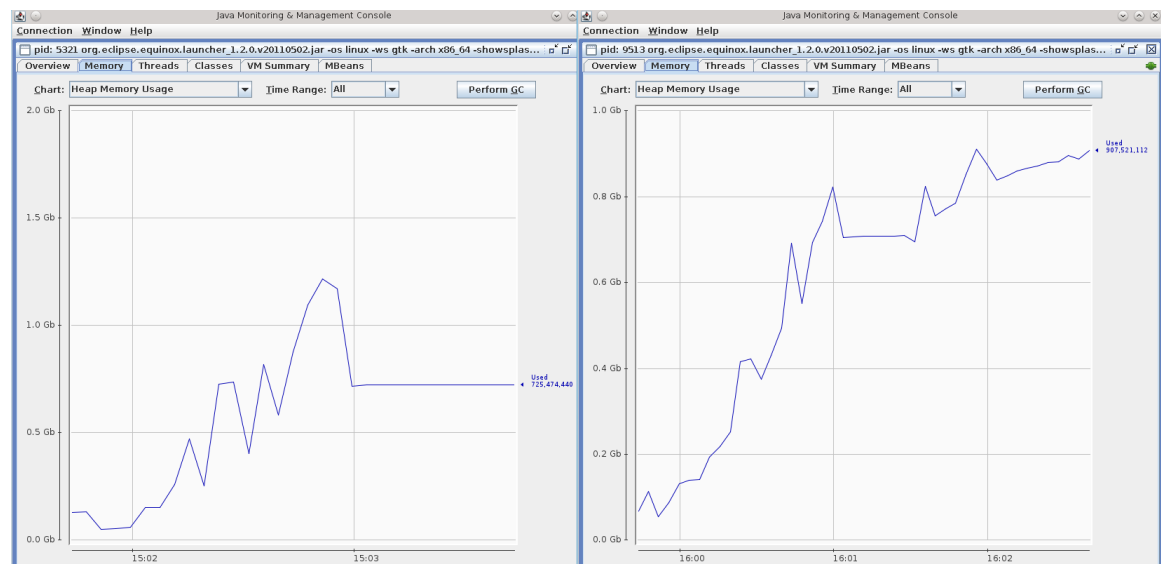


Figure 4.1.: Before the performance improvements - The graph on the left shows the analysis of 346'377 PSLOC with 2048MB virtual memory in Eclipse. The graph on the right shows the same analysis with only 1024MB memory.

4. Metriculator Performance

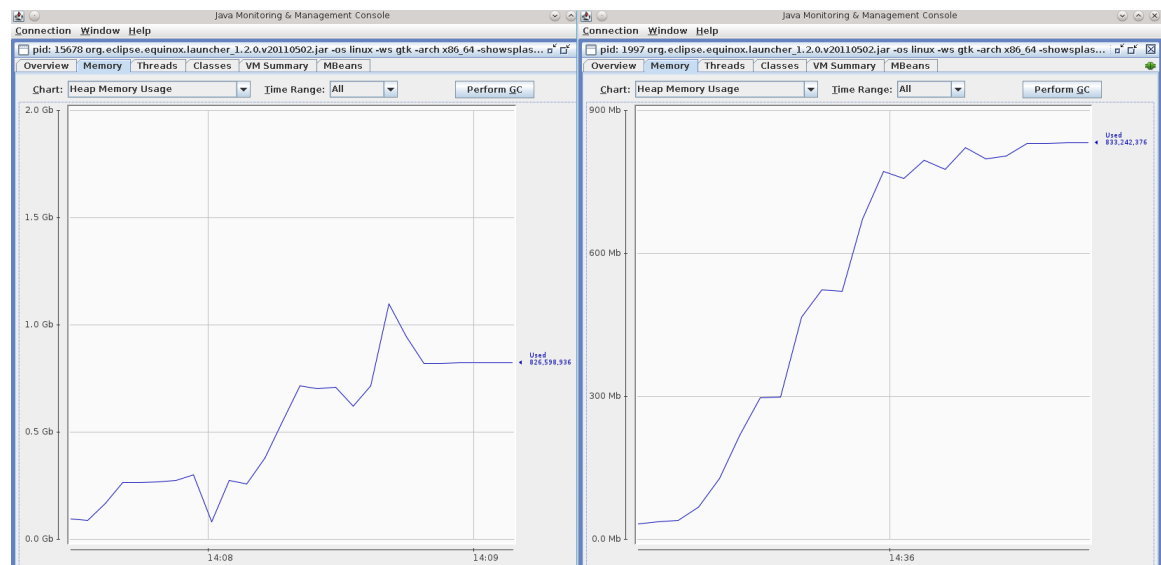


Figure 4.2.: After the performance improvements - The graph on the left shows the analysis of 1'209'742 PSLOC with 2048MB virtual memory in Eclipse. The graph on the right shows the same analysis with only 1024MB memory.

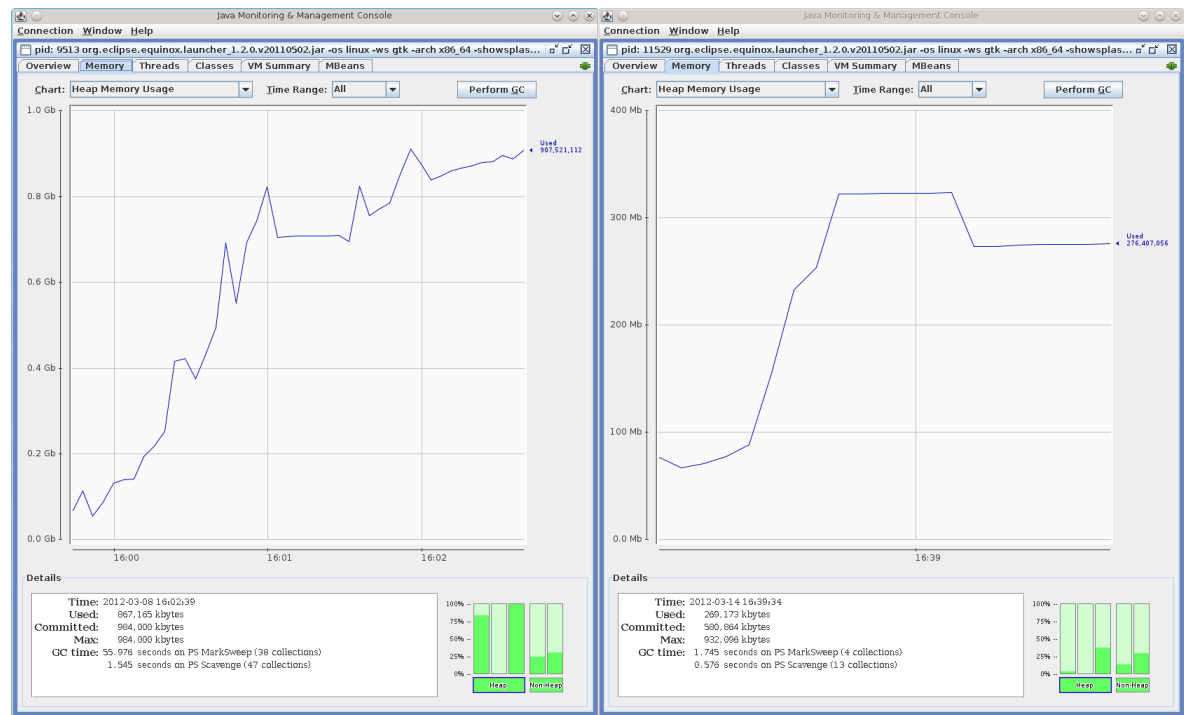


Figure 4.3.: Memory consumption before (left) and after (right) performance improvements, analysing 346'377 PSLOC with 1024MB virtual memory assigned to Eclipse. The analysis on the right is about 4 times faster than the left one and requires only a third of the memory in comparison.

4.2.2. Performance Improvements

This section describes how the performance of metriculator was improved as part of this bachelor thesis. This was mainly achieved by reducing or removing unnecessary references to AST related instances. The NodeInfo refactoring described in section 4.5.2 supported us by identifying and solving performance issues. It simplified the design and algorithms related to AST information management.

4.2.2.1. Problems

metriculator merges declarations with definition. Therefore we must be able to find the a declaration that belongs to a definition. Using bindings provided by the indexer and AST, we can do this. Bindings represent semantic concepts and their properties, and they connect declarations and references. The index of CDT captures all bindings and is built by the indexer, which parses the AST [Sch08]. Thus, it is possible to find matching definitions and declarations with the bindings of the AST nodes.

Some binding instances that we store, reference AST information. Hence, storing binding instances in metriculator may prevent the garbage collector to collect AST instances

that would otherwise be collected. Because we require the bindings to merge nodes in the hybrid tree builder after all checkers ran, we cannot release the bindings resulting in huge memory allocations which impacts the performance. Furthermore, the logical tree is built on demand based on the hybrid tree. To transform the hybrid node structure to a completely logical structure, we still require binding information. See the metRICULATOR documentation [met11a] for further details.

The solution is described in the following paragraphs. The idea is particularly to process the bindings of a translation unit [cpp11, paragraph 2.1.1] and release all references to binding instances after the translation unit has been processed.

4.2.2.2. Merging of Declarations and Definitions in the Hybrid Tree

As of the end of the semester thesis all the binding information of the indexer for the declarations and definitions of functions as well as types were stored in NodeInfo objects associated to the hybrid tree nodes. As soon as a definition of a declaration was found in the same scope during the analysis, the declaration was removed and replaced by the definition. All the bindings have been saved during the static analysis and were also needed after the analysis to build the logical tree.

The merging of declarations and definitions now takes place at the end of a translation unit. After the analysis of a translation unit all the bindings of declarations and definitions in the same scope are collected and resolved. Thus, all bindings of the nodes in a translation unit can be released at the end of the translation unit analysis.

4.2.2.3. Merging of Members in the Logical Tree

The merging of the members of a type is based on the binding that contains the owner hierarchy information. To determine the owner of a node, the binding is used to build a logical owner name. Up to now the logical owner name was built on demand when the logical tree was built. That is why the binding was still needed after the code analysis.

The logical owner name is now built directly after the creation of the node, which means that the binding is no longer needed to build the logical tree.

The concept of string based identifiers for nodes in a tree structure has already been implemented in the hybrid tree using hybrid ids. See metRICULATOR documentation [met11a] section 4.2.2.1. The refactoring allowed us to consequently implement the same concept for logical trees.

4.2.2.4. Merging of Declarations and Definitions in the Logical Tree

Up to now the binding information were located in the node and were used to determine the bindings of declarations and definitions. But now it is no longer possible to get the binding at the time the logical tree is built.

The merging is currently based on the logical owner name and the logical name of the node. The logical owner name determines the scope where the merging takes place and the logical name is used to find the definition for the declaration. Both names were extracted from the binding after the creation of the node.

4.2.2.5. Removing the AST Bindings

The merging of function declarations and definitions is based on index bindings. Merging type declarations and definitions in contrast is based on AST bindings. We completely removed the AST bindings by replacing them with index bindings. These changes caused some errors, which are further described in the open issues section 4.2.3.

This improvement massively decreased the memory consumption as illustrated in figure 4.4, but it did not change the running time of the analysis.

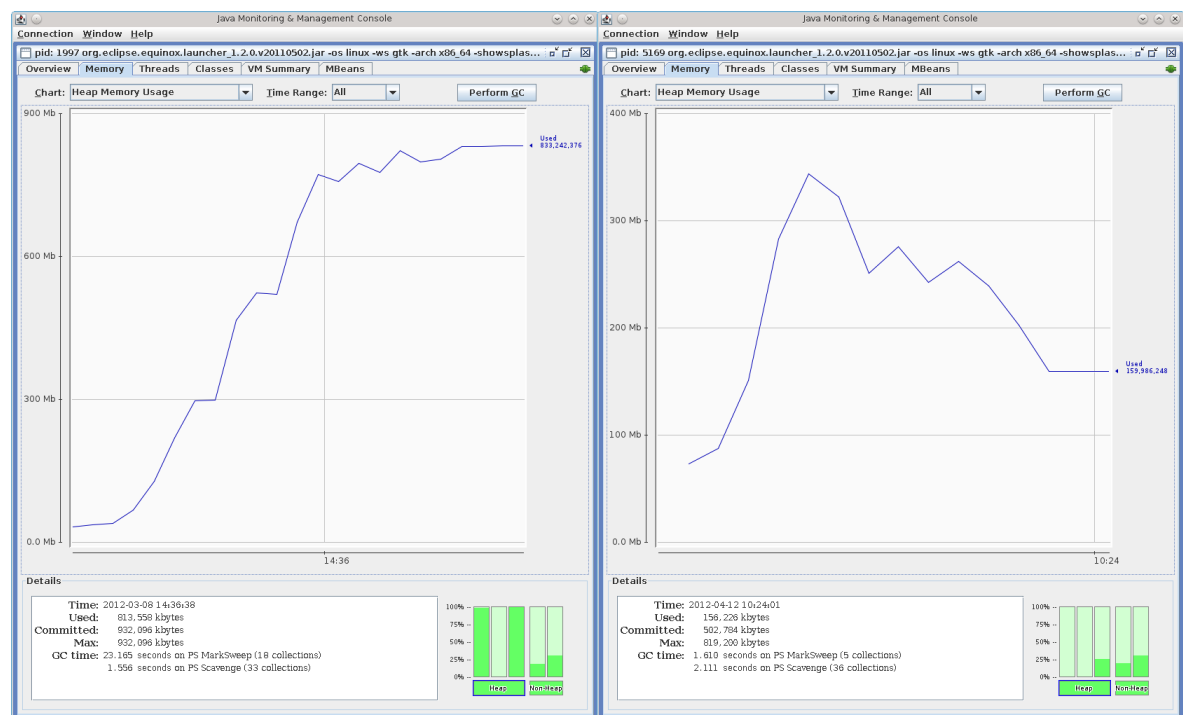


Figure 4.4.: Memory consumption before (left) and after (right) removing the AST bindings. The analysis on the right requires about one third of the memory in comparison.

4.2.3. Open Issues

After the performance has been improved as described in section 4.2.2, we started to further refactor the source code to get rid of the AST bindings entirely. This refactoring further decreased the heap space allocation, hence, analysing doom [doo12] was not a

problem any more (see [met11a] bug #222). Unfortunately, now our tests related to anonymous namespaces failed. We recognised that it is not possible to build the logical owner name based on an index binding the same way we did it before based on an AST binding. This is due to regarding anonymous namespaces, index bindings do not behave the same as AST bindings. AST bindings contain information from the AST and the indexer. The lack of the AST information in index bindings prevented us to generate the logical owner name if anonymous namespaces were in place.

Therefore, instead of building the owner name by ourself, we started to use a helper method from the CDT parser (`CPPVisitor.findNameOwner`) that handles anonymous namespaces correctly. However, using this helper method almost doubled the running time of metriculator. Therefore, we decided to discard the helper method approach for this thesis. Further work is required to support anonymous namespaces while maintaining the current performance. We think that low memory consumption and fast running time is more important than the correct merging of declarations and definitions within anonymous namespaces in the logic tree of metriculator. The hybrid tree is not affected by that decision. It only affects the logical tree nodes hierarchy and values if they contain anonymous namespaces.

4.3. Tag Cloud - Dealing with Large Data Input

The tag cloud component which is based on the cloudio project [zes11] has difficulties to render a tag cloud when there are many words to place in the cloud. More specifically, the bigger the weight span from minimum to maximum weight is, the more frequently it fails. The error is shown to the user. This is not an issue of metriculator but a bug in the tag cloud component. There is no bug report yet, but the author is aware of the bug and will soon fix it. Since the tag cloud component is an optional feature of metriculator, this bug is not critical for this thesis.

4.4. Composite Update Site

We originally intended to set up a composite update site because we wanted to simplify the installation process. Up to now, users had to manually install the zest framework and the CDT, prior to the installation of metriculator. We thought that using a composite update site will resolve that problem in order that Eclipse will be able to automatically resolve all dependencies.

The plug-in separation that moved the tag cloud component as optional feature into its own plug-in, lowered the importance to provide a composite update site. Because the ability to automatically install zest in the same step as installing metriculator was the main motivation to introduce a composite update site. People that want to use metriculator most likely have CDT already installed.

Users that just want to install metriculator can do this by using the normal update site experiencing a fast and simple installation process. The optional tag cloud feature can be installed from the same update site at a later time if desired. However, to install the

tag cloud feature, users still require the zest framework, which has to manually installed in advance.

4.5. Design Changes

The software design of metriculator has been changed for various reasons. Each sub section describes the motivation and consequence of the design changes.

4.5.1. Tag Cloud Extraction

As of the end of the semester thesis the tag cloud component was integrated into the metriculator plug-in. After we published the plug-in, different people proposed to make the tag cloud component an optional feature of metriculator. We have already considered this as well during the semester thesis. Based on our schedule and priorities we decided to not separate it immediately. But now, since we published metriculator in the cdt-dev mailing list and other people shared their opinions about the tag cloud, we decided to make it an optional feature [cdt]. Overall, this decision was taken based on the following issues:

- The zest tag cloud component has some issues with large input data. This issue has already been discovered in the semester thesis. See section 4.3 for more information.
- Before installing metriculator, users have to install the zest framework manually. See [met11a] Appendix C for the installation manual.

4.5.1.1. Component Design

Figure 4.5 illustrates the new design. The tag cloud related classes are moved to a new plug-in so that metriculator does no longer has dependencies on zest. The tag cloud component uses the view menu contribution extension point [vie12] provided by Eclipse to register itself as a contributor of the metriculator view. metriculator automatically detects if the tag cloud component is installed and based on that toggles the tag cloud commands on the user interface.

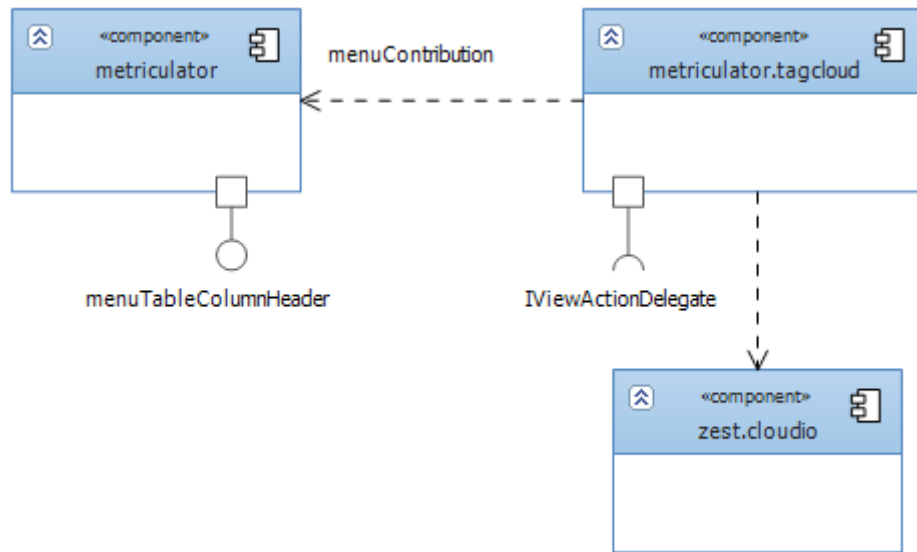


Figure 4.5.: The metriculator component has no dependencies to the zest.cloudio component. The metriculator.tagcloud component registers itself in metriculator to place a command in the table context menu.

4.5.1.2. Consequence

Because the tag cloud component is extracted to a separate feature, metriculator is no more dependent on the zest framework. This simplifies and shortens the installation process of metriculator because users do not have to install the zest framework in advance. The tag cloud component can be installed in a second step if desired. However, before doing so the zest framework has to be installed separately in advance.

4.5.2. NodeInfo Refactoring

This design change is not directly related to a requirement defined in section 4.1. This refactoring changed the `NodeInfo` and `AbstractNode` class hierarchies used in metriculator. Its purpose was to eliminate some design flaws, which are outlined below. The improved design provided us by determining and solving the performance issues described in section 4.2.2.

As of the end of the semester thesis, the AST related information of the classes in the `AbstractNode` composite hierarchy [GHJV95] were encapsulated in the `NodeInfo` class as illustrated in figure 4.6. `NodeInfo` had several constructors, each associated with a specialised node of `AbstractNode`. The specific data and logic of the classes from the `AbstractNode` hierarchy were held and implemented in the large class `NodeInfo` [FM05]. For further information about the `AbstractNode` hierarchy, see the metriculator documentation.

The NodeInfo refactoring was mainly performed in two steps. The first step was about to introduce a NodeInfo hierarchy that resulted in the AbstractNodeInfo hierarchy 4.7. The second step was about to merge the AbstractNodeInfo hierarchy and the AbstractNode hierarchy that resulted in an improved AbstractNode hierarchy 4.8. The motivations and consequences of this refactoring process are described in detail in the following paragraphs.

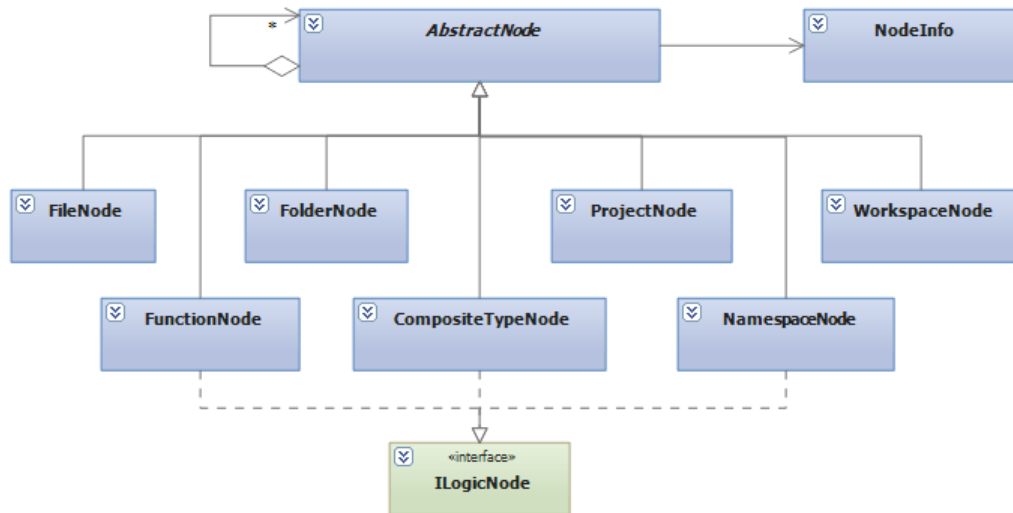


Figure 4.6.: AbstractNode composite hierarchy with AST related information stored in NodeInfo. Every node holds a NodeInfo reference. The logical classes implement the ILogicNode marker interface.

4.5.2.1. Consequences - AbstractNodeInfo Hierarchy

To counteract the data clumps and large class smells [FM05] of NodeInfo we introduced a NodeInfo hierarchy as illustrated in figure 4.7. The specific classes of the AbstractNode hierarchy are now associated with the specific classes of the AbstractNodeInfo hierarchy. This new design approach increases the cohesion of the nodes and its information as well as it improves the separation of concerns. The data and logic of the AbstractNode hierarchy are now encapsulated in the associated AbstractNodeInfo specialisation classes.

This refactoring results in two almost parallel inheritance hierarchies [FM05], the AbstractNode and AbstractNodeInfo hierarchy with the following associated classes:

The remaining classes of the AbstractNode hierarchy do not have any AST related information and thus no associated classes in the AbstractNodeInfo hierarchy.

AbstractNode	AbstractNodeInfo
AbstractNode	AbstractNodeInfo
FileNode	FileSystemNodeInfo
FolderNode	
ILogicNode	LogicalNodeInfo
FunctionNode	FuncDeclNodeInfo FuncDefNodeInfo
CompositeTypeNode	TypeDeclNodeInfo TypeDefNodeInfo
NamespaceNode	NamespaceNodeInfo

Table 4.2.: Class association mismatch between AbstractNode and AbstractNodeInfo hierarchy.

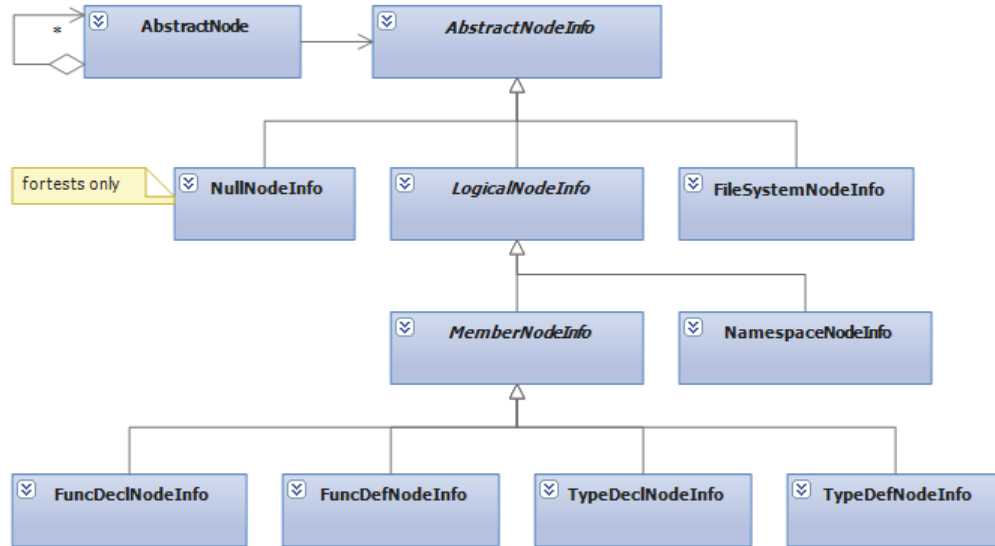


Figure 4.7.: AbstractNodeInfo hierarchy with AST related information distributed in specialised NodeInfo classes. Every class of the AbstractNode hierarchy is associated with a class of the AbstractNodeInfo hierarchy. The abstract class LogicalNodeInfo contains the shared logic and data of the logical AST information. The abstract class MemberNodeInfo contains the shared logic and data of the specific function and type information of the AST. The NullNodeInfo serves for test purposes only, where no AST infos are required.

As already mentioned, the AbstractNodeInfo classes encapsulate the AST related information of the specific classes of the AbstractNode hierarchy. Each AbstractNode derivative uses a specialised AbstractNodeInfo instance to manage AST related information. As depicted in the Table 4.2, some AbstractNode derivatives relate to more than one AbstractNodeInfo specialisation. This is because up to now the AbstractNode

hierarchy does not reflect the AST hierarchy of CDT but the `AbstractNodeInfo` hierarchy does. This inconsistency in design complicates the usage of `AbstractNode` instances. For instance, to distinguish whether an `AbstractNode` instance represents a function declaration or function definition node, we first need to get the associated `AbstractNodeInfo` instance of the `AbstractNode`.

4.5.2.2. Consequences - Improved AbstractNode Hierarchy

To improve the quality of the design of metRICulator and its `AbstractNode` hierarchy we restructured the `AbstractNode` hierarchy as illustrated in figure 4.8. This improved `AbstractNode` hierarchy allows to merge the whole `AbstractNodeInfo` 4.7 hierarchy into the `AbstractNode` hierarchy and thus increases the cohesion of the `AbstractNode` classes and its AST related information.

The AST related information are now placed directly into the `AbstractNode` and its specialised classes which makes the `AbstractNodeInfo` classes needless. A further benefit of the merged hierarchies is that the `NullNodeInfo` [GHJV95] is no longer needed because all `AbstractNode` classes can be instantiated without any AST related information required. This simplifies unit testing because not all tests require AST information, for instance model and tree builder tests.

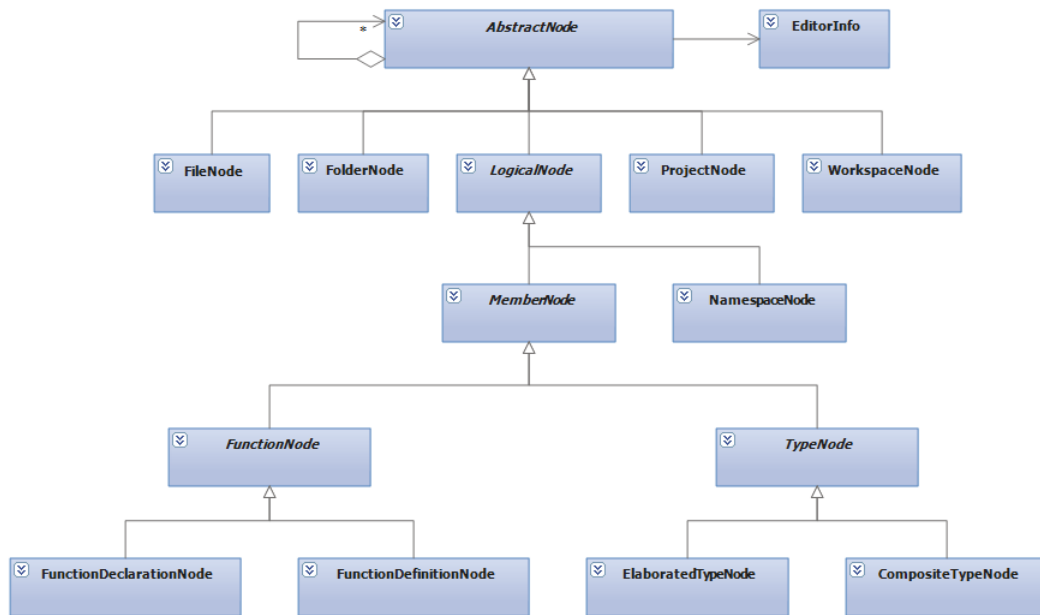


Figure 4.8.: Improved AbstractNode composite hierarchy - The abstract classes (AbstractNode, LogicalNode, MemberNode, FunctionNode, TypeNode) of this hierarchy contain the shared logic and data of its specialised classes. The other classes represent the specific classes of the AST, except of the EditorInfo class. EditorInfo encapsulates information of the AbstractNodeInfo class required by Eclipse to show the node location in a source code editor and information to required to report problems.

4.5.2.3. Consequences - Improved AbstractNode Hierarchy - Merging Details

Looking at the differences between the improved AbstractNode hierarchy and the late AbstractNode hierarchy (figure 4.9) with its associated AbstractNodeInfo classes, we see that AbstractNodeInfo has a lot of methods that should have been moved to specialised classes. Since we merged the AbstractNodeInfo implementation into the AbstractNode class it is now possible to move all AbstractNodeInfo methods to specialised AbstractNode classes, as illustrated in the figures 4.9 and 4.10. Most of the data and logic of AbstractNodeInfo were pushed down in the AbstractNode hierarchy to the classes where they belong to, as illustrated in the class diagram in figure 4.11 showing only classes that represent logical nodes.



Figure 4.9.: AbstractNodeInfo hierarchy - This figure shows the upper part of the hierarchy without the specialisations of LogicalNodeInfo, see figure 4.7 for the whole class diagram. The specialised classes of AbstractNodeInfo override the default implementation of AbstractNodeInfo if required. Therefore, the specialised classes often refuse their bequest [FM05] which is not a very good design and leads to ugly and confusing code.

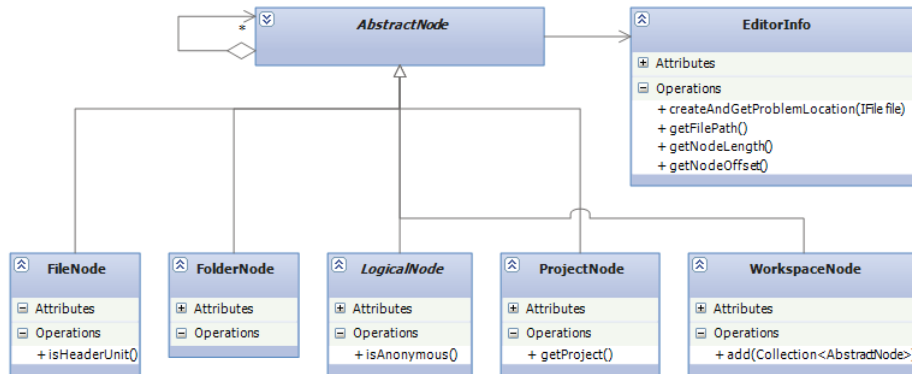


Figure 4.10.: AbstractNode hierarchy - This figure shows the upper part of the hierarchy without the specialisations of LogicalNode, see figure 4.8 for the whole class diagram. In comparison to the class diagram of the AbstractNode-Info hierarchy in figure 4.9, these classes only implement what they are responsible for. There is no unnecessary default implementation neither a NullNodeInfo class for test purposes.

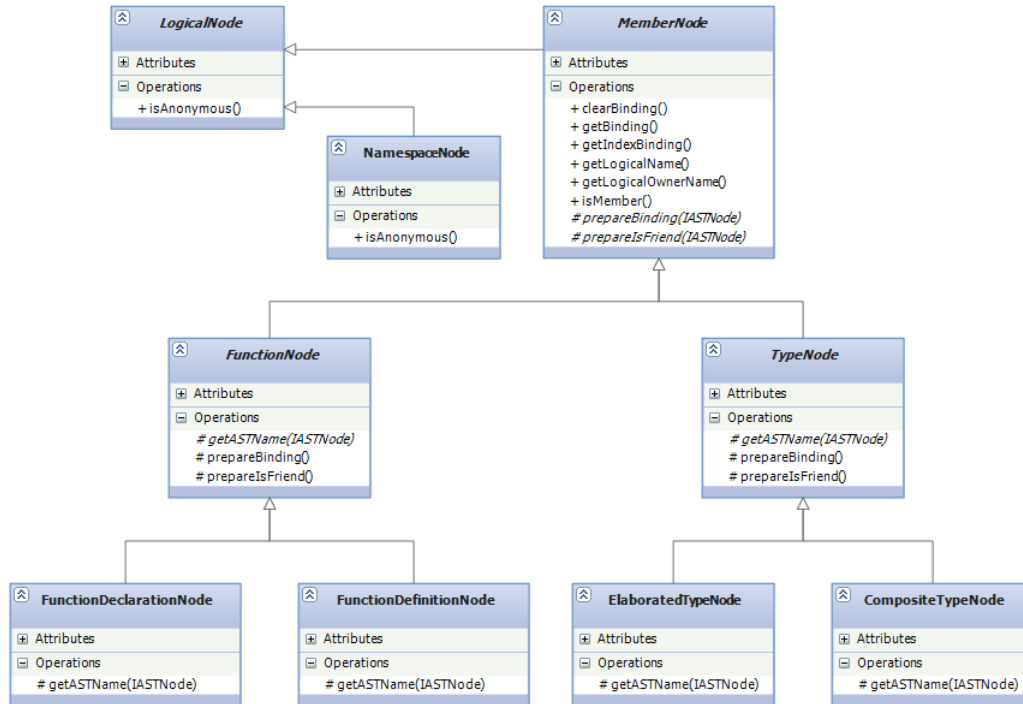


Figure 4.11.: AbstractNode hierarchy - This figure shows the lower part of the hierarchy with the logical nodes. As already mentioned in figure 4.10 it is now a proper design.

4.6. Further Improvements

This section describes further improvements that were applied to metriculator. They are not part of the objectives or requirements but refer to open issues that will be fixed at a later time. Each improvement is associated to an issue number in our Redmine project [met11a].

4.6.1. GUI Guidelines

As part of the preparation to commit metriculator to the CDT (#180), we checked the Eclipse guidelines [ecl12] and figured out that metriculator does not have a view menu that contains all commands from its toolbar, guideline 6.10, issue #195.

To fix this we simply added a view menu and added the commands of the toolbar also to the view menu.

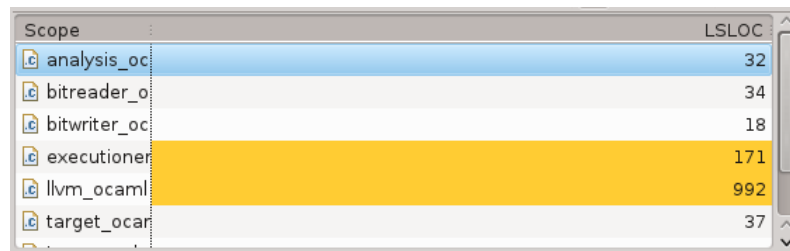
4.6.2. Minor Bug Fixing

The following bugs were fixed. They have less priority since they are not subject of the requirements.

Extra Column At the End Issue #116. metriculator displays the metric values in columns.

Each metric has its own column. Depending on the view mode the jface TreeViewer or TableViewer component is used [jfa12]. On Unix based systems these components stretch the last column to fill the remaining space of the control, in contrast to Windows based systems where the last column behaves the same as all other columns. On large screens the metric values in the last column are separated confusingly far right as illustrated in figure 4.12.

To overcome this issue, we added one extra column at the end that automatically fills up the remaining horizontal space.



Scope	LSLOC
analysis_oc	32
bitreader_oc	34
bitwriter_oc	18
executioner	171
llvm_ocaml	992
target_ocaml	37

Figure 4.12.: The jface TreeViewer and TableViewer components stretch the last column to fill the remaining horizontal space.

Number of Rows Issue #211. This is a small enhancement contributed to the filter views. The first column header shows the number of rows displayed in the table. This allows the user to easily see the number of analysed files, functions, types or namespaces.

4.7. Unit Testing

The unit tests can be divided into three categories: model, tag cloud and checker test cases. The model and checker test cases are part of the test suite that is executed by the continuous integration server (CI-server) on every push to the VCS. The checker test cases require the CI-server to support headless builds. Headless builds run without an Eclipse UI. This build system is provided by the Eclipse PDE (Plug-in Development Environment) in conjunction with Maven [mav11]. Read more about the test set up in the metriculator documentation [met11a, Appendix A.6.2].

Model Tests to verify that the nodes, tree builders and tree visitors work correctly. The tree builder tests rely on the test infrastructure provided by Codan. See section 4.7.1 for further information. All other model tests do neither require Eclipse nor Codan to run.

Checker Tests to verify that the checkers of metriculator produce correct metric values.

The checker tests require the Codan test infrastructure to run. See section 4.7.1 and 4.7.2 for further information.

Tag Cloud The tag cloud tests have no assertion rules at all. Neither are they part of the test suite. They just simplify to start and test the tag cloud component within Eclipse.

For further details about unit testing and test coverage see the metriculator documentation [met11a, Appendix B.4].

4.7.1. Codan Test Infrastructure

Codan provides a test infrastructure that allows each test method to be fed with a piece of C++ source code. The source code is placed above the test method as comment. This feature is available by extending the `CodanTestCase` class.

Codan offers different ways to process the source code in comment. We use two different variants. One variant loads the code without indexing it, the other loads the code and runs the indexer. Both variants are conceptionally illustrated in the listings 4.2 and 4.3 respectively.

```
/* code to be tested */
public void testXXX(){
    loadCodeAndRun(getAboveComment()); // runs the code above without
                                        // building the Index

    /* test logic */
}
```

Listing 4.2: Sample test method that does not run indexer on method comment.

```
/* code to be tested */
public void testXXX(){
    loadcode(getAboveComment());
    runOnProject(); // runs the code above with building the Index

    /* test logic */
}
```

Listing 4.3: Sample test method that runs indexer on method comment.

4.7.2. Checker Tests

For checker test cases, we extend the `CheckerTestCase` class, so that the source code in comment is analysed by the checker under test. Listing 4.4 shows a real world checker test method used in metriculator.

```
// #if a<0
// #endif
public void testPreprocessorIfStatement(){
    loadCodeAndRun(getAboveComment());

    assertEquals(2, workspaceNode.getValueOf(metric).aggregatedValue);
}
```

Listing 4.4: Checker test method that verifies the McCabe value calculated by the McCabeMetricChecker. The workspaceNode is an AbstractNode instance that is the root of the hybrid tree builder, created in the setUp-method.

4.7.3. Indexer Based Tests

Some model tests require indexer information to work correctly, e.g. merging of types and functions. Codan provides a way to run the indexer on the source code in method comments (Listing 4.3). Listing 4.5 shows a real world example of an indexer based test used in metriculator. For further information about function merging see section 4.2.2.

```
// namespace Outer { // at depth 0
//     namespace {
//         struct A {
//             void fx();
//         };
//         void A::fx(){}
//     }
// }
public void testNestedAnonymousNamespaceMemberMerging(){
    loadcode(getAboveComment());
    runOnProject();

    root = MetriculatorPluginActivator.getDefault().getLogicTreeBuilder().root;

    // only Outer present
    assertEquals(1, root.getChildren().size());
    // only struct A present
    assertEquals(1, getFirstChildInDepth(root, 1).getChildren().size());
    // only one child in A present
    assertEquals(1, getFirstChildInDepth(root, 2).getChildren().size());
    // the only child in A is a function definition
    assertTrue(getFirstChildInDepth(root, 3) instanceof FunctionDefNode);
}
```

Listing 4.5: Real world example test method that runs indexer on method comment. The indexer information is required to replace the declaration of fx() with its definition in the logic tree.

he idea behind namespactor is to provide a tool that simplifies and accelerates working with names

Statement of Authorship

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass ich/wir keine durch Copyright geschützten Materialien (z.B. Bilder, Messdaten) in dieser Arbeit in unerlaubter Weise genutzt habe(n).

Ort, Datum:

Ueli Kunz, Name, Unterschrift:

Julius Weder, Name, Unterschrift:

A. Environment Set up

This appendix describes the hardware and software components that support us in reaching our project goals. We give detailed installation and configuration instructions and highlight problem areas to be aware of when setting up a similar environment.

Where possible we use the same set up in this bachelor thesis as we used in the semester thesis [met11a].

A.1. Hardware

We use a virtual server to host different kinds of software that support us in our daily project tasks. The virtual server is hosted by the HSR. We have full root access and can connect to the server by VPN if we are outside of the *HSR-LAN*. The server runs with Ubuntu 10.04 TLS on 1GB RAM. The host name is `sinv-56013.edu.hsr.ch`.

A.2. Project Management Software

Our Redmine instance is publicly and read only available at <http://tiny.cc/namespaceactor>. For further details see [met11a].

A.3. Version Control System, Git

To support our file version management we decided to use Git. The latest release at the start of our project was version 1.7.6. For further details see [met11a].

A.4. Development Environment

The plug-in was developed in Eclipse Indigo using the plug-in development environment (PDE) plug-in. To test namespaceactor we use self written C++11 source code and the CDTTesting framework [Fel12]. To force the compiler to build according to the C++11 standard add the `-std=gnu++0x` flag to the following field: *Project Properties* > *C/C++ Build* > *Settings* > *GCC C++ Compiler* > *Miscellaneous* > *Other flags*.

A.5. Build and Deployment Automation

The CDT project supports ant and Maven as build automation platform. We use Maven (in contribution with Tycho version 0.14.1) because it seemed a lot easier to maintain than ant and has already been used in recent projects at HSR as well as in some CDT projects.

The metriculator plug-ins are deployed as nightly and stable builds. namespactor is only available as nightly build because it has not been made public yet. Take a look at our Jenkins server to access the builds [jen12].

A.5.1. Maven XML Configuration

Maven uses pom.xml for build instructions. We have one pom.xml in the root directory of all Eclipse projects (root pom) and one in each subdirectory (project pom). All pom files are checked-in to the VCS as well. The pom files used for namespactor are very similar to the ones used in the semester project, see [met11a] for more details.

A.6. Testing Eclipse CDT Refactoring Plug-ins

When developing a C++ refactoring plug-in, the refactoring can be manually tested in Eclipse. This is a very time consuming and inefficient approach. To follow a test driven development cycle [Bec03] it is best to have a tool that allows to automatically execute tests, this also applies to refactorings. Our test suite is built upon the recently announced CDTesting plug-in created by the IFS [Fel12].

A.6.1. CDTesting Framework

The CDTesting plug-in simplifies writing and running JUnit4 tests for CDT plug-ins. It is based on the RTS framework that has already been used in other theses to create automated refactoring tests (e.g. [MI10]). The RTS framework is integrated into CDT. A good description of it can be found at [MI10]. The CDTesting Eclipse plug-in is available from <http://dev.ifs.hsr.ch/updatesites/cdttesting/>. The set up of the plug-in is described in appendix C.

A.6.1.1. Plug-in Concepts

This paragraph explains the features of the CDTesting plug-in which encapsulates the RTS framework and how the plug-in is intended to be used. In contrast to plain RTS tests the plug-in provides the following benefits:

- Parametrised JUnit4 test cases that allow to re run single unit tests (i.e. RTS test cases)

- Complete set up of the CDT index prior to test case run
- External include paths (e.g. the STL is by default not available in CDT tests)
- Dependencies to other projects and files in other projects

The diagram in figure A.2 illustrates the basic concepts in a CDTTesting plug-in environment. A JUnit4 test suite class may be used to run multiple JUnit4RTSTest test cases. The CDTTesting plug-in associates each Java test case class with one RTS file. By default the RTS file of a test case class is looked up at predefined location. The default location is defined in listing A.1 and illustrated by an example project in figure A.1.

```
<plug-in_root>/<extension-point_sourceLocation>/ \
<test-case-class_package-appendix>/<test-case-class_name>.rts
```

Listing A.1: Default lookup location of an RTS file by a given test case class. Where *extension-point_sourceLocation* is the extension points property value defined in the plug-in xml and *test-class_package-appendix* is the fully qualified package name of the test case class trimmed by the fully qualified package name of the activator class of the test project.

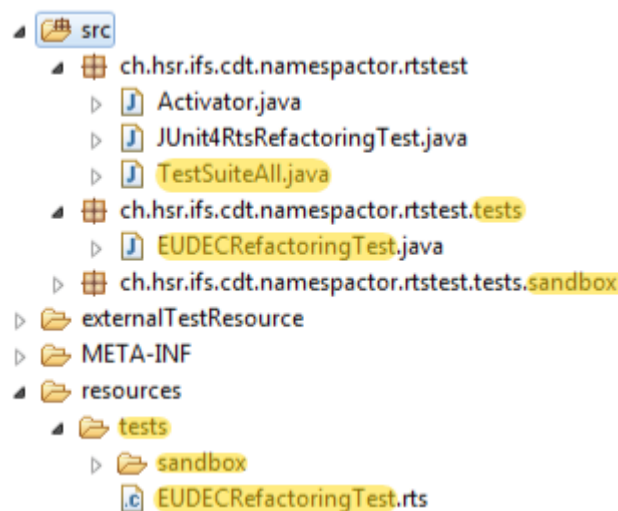


Figure A.1.: Eclipse test project using the CDTTesting plug-in. The name of the RTS file is the same as the class name of the JUnit4RtsTest test class. The path of the RTS file is derived from the package name of the test class as illustrated by the highlighted names.

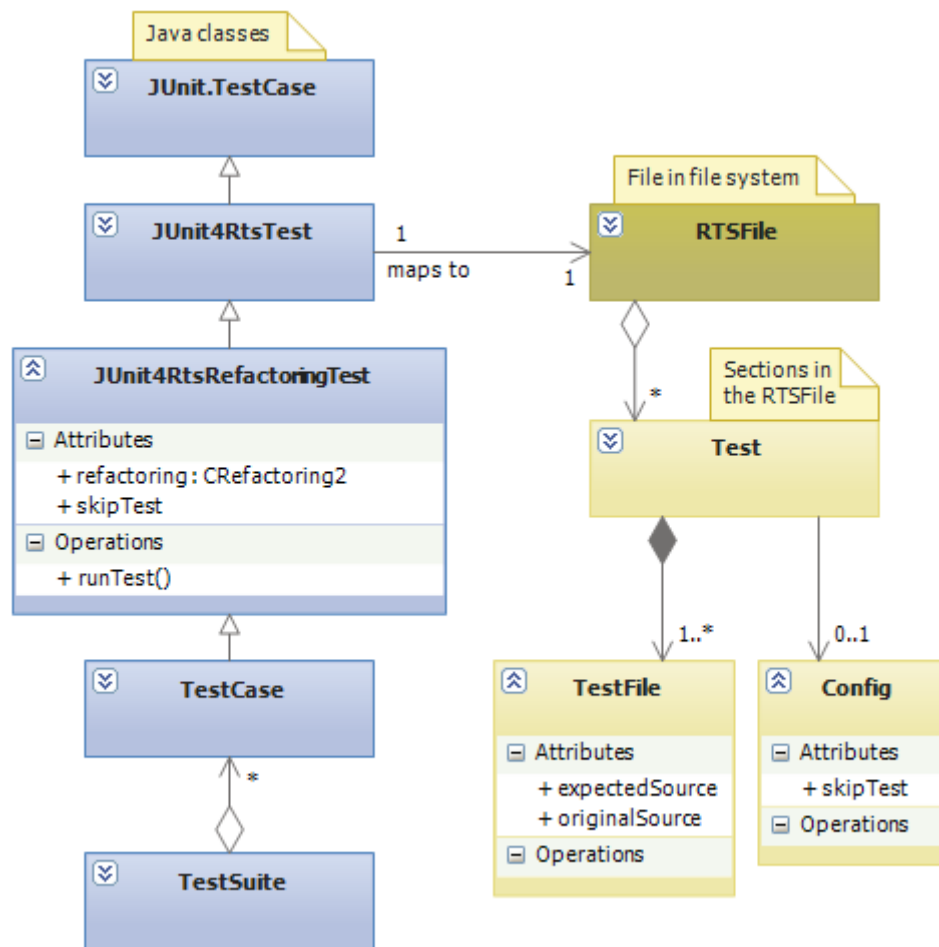


Figure A.2.: Concepts related to the CDTTesting plug-in and the RTS framework respectively.

RTS files have an own syntax. The syntax is described at [MI10]. Figure A.3 shows how the abstract concepts of figure A.2 map to a sample RTS file. A RTS file contains test case definitions. Each test case defines one or more test files that are part of the test. A test file is divided into two sections: the expected source and the original source section. The expected source section is optional for files not expected to be changed by the refactoring.

A test case can have a config section with custom attributes. For example, in namespactor we introduced the attribute *skipTest* to indicate that JUnit4 should not run the test. This way we can commit draft versions of test cases or test cases for bug resolutions that are in progress without producing build failures.

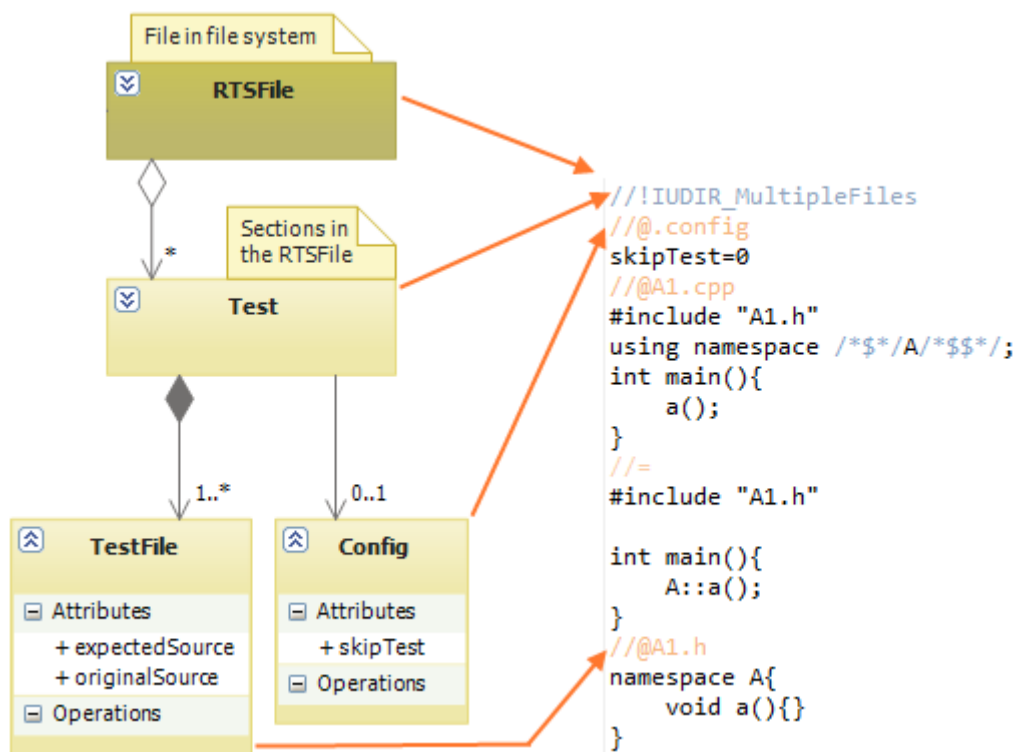


Figure A.3.: Concepts of the RTS framework mapped to a RTS file.

A.6.1.2. RTS Editor Outline

There is a special editor for RTS files. The editor does syntax highlighting and features an outline view. The editor can be downloaded from http://sinv-56013.edu.hsr.ch/rts_editor.zip. To install, just extract the zip into the root folder of your Eclipse installation.

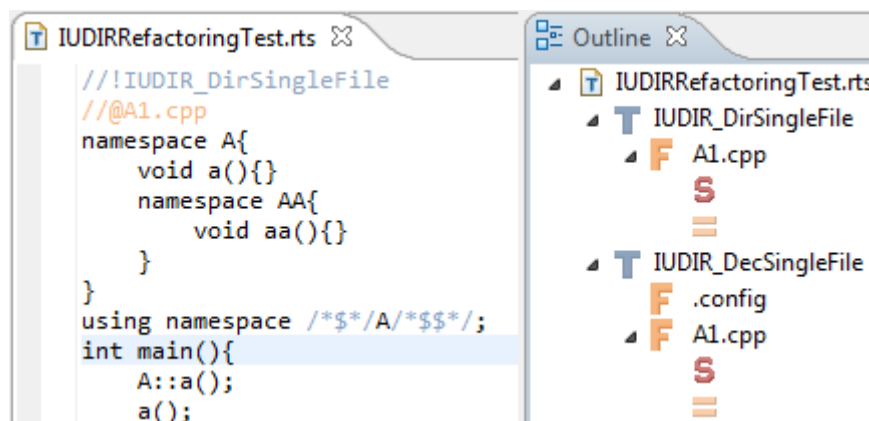


Figure A.4.: Screenshot of the RTS editor and its outline view.

A.6.1.2.1. RTS Test File and Selections In RTS tests, the text selection (defined by `/*$*/.../*$*/`) has to be in the first RTS test file of a RTS test. The RTS API provides access to an `ITranslationUnit` instance. In test mode this is always linked to the first file in your RTS test definition. Therefore, to get the selection and its underlying AST node you have to place the file with the selection on top of your RTS test (after the optional config section).

A.7. AST Rewrite Store

The AST rewrite store is the name of a set of classes that simplifies creating AST rewrites. The `ASTRewriteStore` wrapper class allows to add insert, remove and replace changes. The class stores all rewrites created during a refactoring process and selects the appropriate one for the current change, by considering the rewrite root of a change. If a change creates a new rewrite, it is saved inside the store for further usage.

Although the AST rewrite store simplifies creating AST rewrites a lot, some limitations still exist. For instance, nested AST rewrites are not possible. But this is not a limitation of the AST rewrite store but the underlying framework. For more information read the documentation of the author at [tur] section 4.3.1.

A.8. DOM AST View

The DOM AST view is an Eclipse view that outlines the AST of the active document in a tree view (figure A.5). It was first announced by Devin Steffler in the `cdt-dev` mailing list in 2005 [Ste05]. Because there is no update site, we published the plug-in at <http://sinv-56013.edu.hsr.ch/DOMASTView.zip>. To install, extract the zip into your Eclipse workspace root and add the location `${workspace_loc}/testing-project` to your target definition.

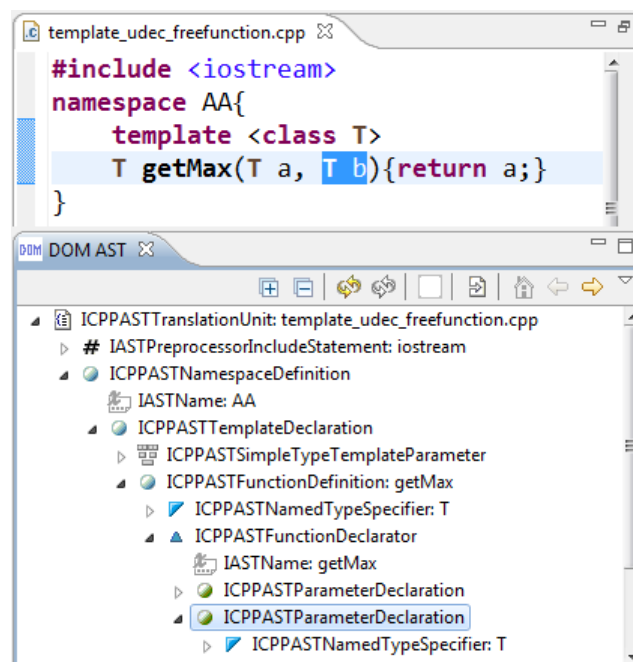


Figure A.5.: Screenshot of the DOM AST view.

B. Terminology

This terminology mainly applies to namespace related content. As long as not otherwise noted, this thesis respects the grammar of the C++11 standard [cpp11].

- For better understanding and to prevent confusion, listing B.1 illustrates the grammar description of the C++11 standard for namespace specifiers.
- To simplify reading we introduce the term *name specifier* that is used instead of *nested-name-specifier* or *namespace-name*.
- A qualified name is called fully qualified if it starts with the scope resolution operator `::`.
- The C++11 standard distinguishes between *name* and *identifier* although they mean almost the same: *identifier* is used for describing the grammar of the language and *name* is used in descriptions where no grammar is included. We use the term *name*.
- The term *namespace using declaration* is shorten to using declaration, since this thesis focuses on *namespace using declarations*, not *class member using declarations*.

```
namespace A {  
    namespace B {  
        void f();  
    };  
};  
  
//A          unqualified-id -> unqualified  
//::A        no nested-name-specifier -> fully qualified  
//A::B       namespace-name: A:: -> qualified  
//::A::B     namespace-name: A:: -> fully qualified  
//A::B::f    nested-name-specifier: A::, namespace-name: B:: -> qualified  
//::A::B::f  nested-name-specifier: A::, namespace-name: B:: -> f. qualified
```

Listing B.1: Terminology of names often used in the analysis section of the namespace chapter.

C. CDTTesting Plug-in Set up

C.1. Quick Start

To set up the CDTTesting plug-in [Fel12] to get started quickly you can just import the existing example project:

1. Add <http://dev.ifs.hsr.ch/updatesites/cdttesting/> as location to your target definition. Update the location and reset the target platform.
2. Download the zip from the CDTTesting source repository [Fel12].
3. Create a new empty Java project.
4. Extract the zip to any folder.
5. In Eclipse, choose File > Import > Plug-ins and Fragments. Import from the directory you just extracted to. Import as *Project with source folders*.
6. Add the example project to be imported and start the import.

C.2. Set up for Refactoring Tests

To set up the CDTTesting plug-in [Fel12] to test refactorings follow these steps:

1. Add <http://dev.ifs.hsr.ch/updatesites/cdttesting/> as location to your target definition. Update the location and reset the target platform.
2. Download the zip from the CDTTesting source repository [Fel12].
3. Create a new empty Java project.
4. Paste the folder *externalTestResource* from the downloaded zip into the test project root.
5. Create a class that serves as base for all refactoring test cases. Place it in the root package (e.g. `ch.hsr.ifs.cdt.namespactor.rtstest`). It extends the `JUnit4RtsTest` base test case. Since the CDTTesting plug-in is designed to help building test suites for CDT plug-ins in general, this extension is required to create a specialised refactoring test case base class. The source is available at [nam].
6. Add a test suite class to the project (e.g. in package `ch.hsr.ifs.cdt.namespactor.rtstest`). That is just a normal Java class annotated with `@RunWith(Suite.class)` and `@SuiteClasses(Class<?>...)`.

7. Create a package for your Java test classes (e.g. `ch.hsr.ifs.cdt.namespactor.rtstest.tests`).
8. Add the following xml to the `plugin.xml` of your test project:

```
<extension point="ch.hsr.ifs.cdttesting.testingPlugin">
  <testResouresLocation sourceLocation="/resources/" \
    activatorClass="ch.hsr.ifs.cdt.namespactor.rtstest.Activator"/>
</extension>
```

Listing C.1: Registration at the extension point of the CDTTesting plug-in.

sourceLocation is the project relative path to the root directory where your RTS files reside.

activatorClass is the fully qualified name of the plug-in activator class of your test project.

9. Create the folder *resources* (previously specified as `sourceLocation`) in your project root.
10. Create a sub folder called *tests* (according to the package name suffix where the test classes reside).
11. Create a test class *SampleTest*, that extends the created test case base class, in the package `ch.hsr.ifs.cdt.namespactor.rtstest.tests`.
12. Create a RTS file at */resources/tests/SamplteTest.rts*

Now you are ready to define RTS tests in the created *SampleTest.rts* file and run the test with JUnit4 by running the created test suite.

D. IUDIR Refactoring - Indexer Implementation

The IUDIR implementation has the most complex algorithm of all inline refactorings. The IUDEC refactoring is similar to the IUDIR refactoring, hence, both follow the same basic algorithm, which is shown as pseudo code in listing D.1. The QUN refactoring is much more simpler, because only one name is affected, see section 3.3.6.6 for more details.

This chapter describes the outdated IUDIR refactoring implementation using an indexer lookup algorithm, see section 3.3.5 for more details on the different lookup algorithms. For a description of the new hybrid based IUDIR refactoring implementation see section D.

```
selectedUsing      = getSelectedUsing(selection);
sourceDeclaration = getDeclarationOf(selectedUsing); // e.g. namespace def.
usingScope = getScopeOf(selectedUsing);
candidates = findReferencesOf(sourceDeclaration);
candidates = filterByScope(candidates, usingScope);
candidates = filterValid(candidates);
inlineDeclarationName(candidates, sourceDeclaration);
```

Listing D.1: Pseudo code for the IUDIR and IUDEC algorithms.

The sequence diagram in figure D.1 summarises the `checkInitialConditions` algorithm of the IUDIR refactoring. IUDEC and QUN only implement sub steps of this algorithm, see 3.3.6.5 and 3.3.6.6 respectively. Below, each method shown in the sequence diagram is explained.

checkInitialConditions This method is invoked by the LTK framework prior to show the refactoring wizard. If the returned refactoring status contains an error the wizard is not shown, instead the error is displayed to the user. See section 3.3.2 for more details.

getSelectedUsing Gets the selected using directive. It uses a visitor to visit all declarations within the active `IASTTranslationUnit` and returns the one that is within the offsets of the active text selection provided by Eclipse. If null is returned, no using directive is selected and the user is notified appropriately.

findCompoundStatementInAncestors Recursively traverses the AST upwards, starting at the selected using directive, until an `IASTCompoundStatement` is found, if not, null is returned. The compound statement is required to validate if a candidate reference node is in the same scope as the originating using directive where the

refactoring was initiated from. See method `isChildValid`. If no compound is returned, the `IASTTranslationUnit` is used as scope.

findChildrenBindingsRecursive The insides of this method are described in detail in appendix D.1. It basically collects all declaration bindings within the namespace referred to by the selected using directive.

processNamespaceDefinitions For each `IASTNamespaceDefiniton` in the map built in the `findChildrenBindingsRecursive` method (see D.1), this method sets the active `NamespaceInlineContext` (figure D.2) and calls `processNamespaceChildren`.

processNamespaceChildren This method simply iterates over all children bindings of the active `NamespaceInlineContext` (figure D.2) and invokes the `processChildReferences` method for each child.

processChildReferences All references of the given child are iterated over and validated. If valid, the child reference is handed over to the method `processReplaceOf`.

isValidChild This method validates the reference instance. It returns true if the name of the referencing node should be changed. A node is invalid if it is:

- an implicit operator call. See section D.2.1 for more details.
- or not within the scope of the selected using directive. See method `findCompoundStatementInAncestors`.
- or already qualified with a trailing name of the using directive. E.g. inlining `A::B` on a name that is already qualified with `B` is not required.

getNodeOf This method returns an `IASTNode` instance for a given `IIndexName`. This method is associated with an open issue, see 3.4.1 for more details.

processReplaceOf Creates a replace change and a remove change in the `ASTRewriteStore` (see appendix A.7 for more information). The given `IASTName` instance will be replaced by its qualified version and the selected using directive will be removed. See section 3.3.4 for more details on how names are built.

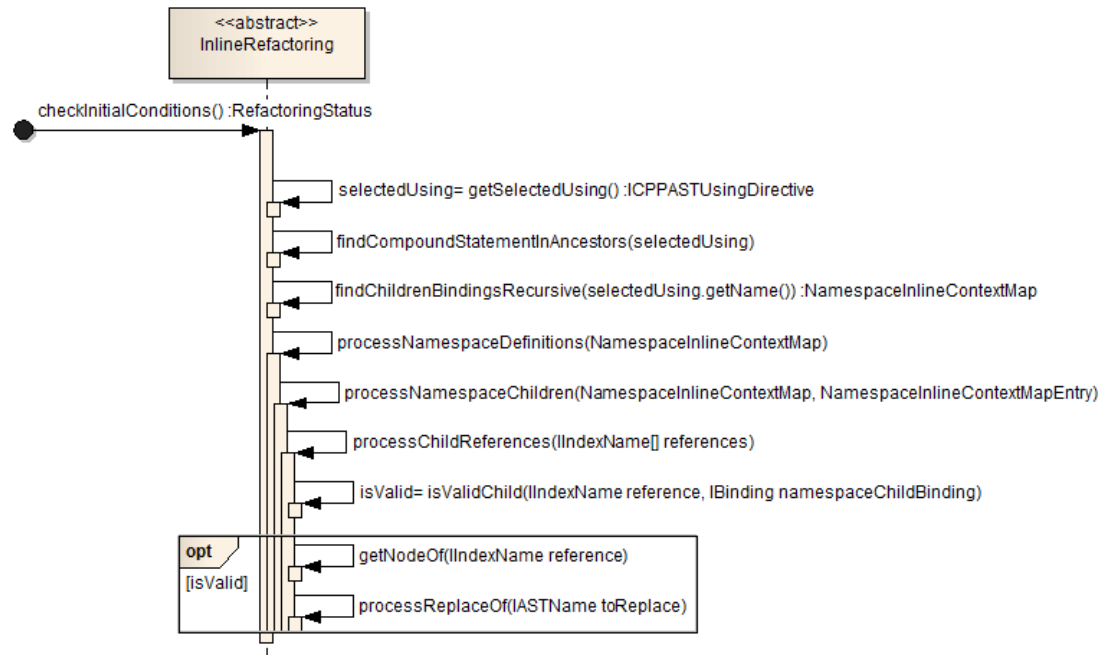


Figure D.1.: Sequence diagram illustrating the IUDIR `checkInitialConditions` implementation. The `checkInitialConditions` method is called from the LTK framework before the refactoring wizard is shown. See section 3.3.2 for more details on LTK.

D.1. Finding References Recursively

The following explanation relates to the code in listing D.2. A using directive (A) introduces all names of a namespace into the scope where it is placed in. If the introduced namespace contains another using directive (B) these names are also introduced. On the other hand, if using directive A is removed, all the names introduced by it are no longer valid without qualification, this also applies to names introduced by the nested using directive B. Therefore, when inlining using directive A, also the names, in the scope of using directive A, introduced by using directive B, must be qualified. See paragraph 3.1.3.4.4 in the Analysis section for more on nested using directives.

To find all potential names that must be inlined, the IUDIR algorithm works recursively. In the AST, all children of a namespace definition are declarations. First, all declaration names in the namespace definition, referred to by the selected using directive, are collected as potential names. The same is recursively repeated for any using directive that is a direct child of the referenced namespace definition. These names are the output of the algorithm (see method `findChildrenBindingsRecursive` in figure D.1). They will be further processed in a next step, where all references of all names are collected. These references are then filtered and will finally be qualified if required. For example, one filter is the indexer scope filter, see paragraph 3.3.6.7.1 for more details.

The recursive search algorithm uses a context object with information about the cur-

rently searched scope, i.e. a namespace definition, and a collecting parameter object to store all found names. Both objects are encapsulated within the `NamespaceInlineContext` class, see figure D.2.

One namespace (name) can be defined multiple times within the same translation unit, all definitions having the same `IBinding`. For instance, namespace A may be defined in file A.cpp and in file B.cpp. The recursive search algorithm collects the declarations in both definitions. To gather all declarations the algorithm uses a map (`NamespaceInlineContextMap`) with the key being a namespace definition and the value a `NamespaceInlineContext`, see figure D.2.

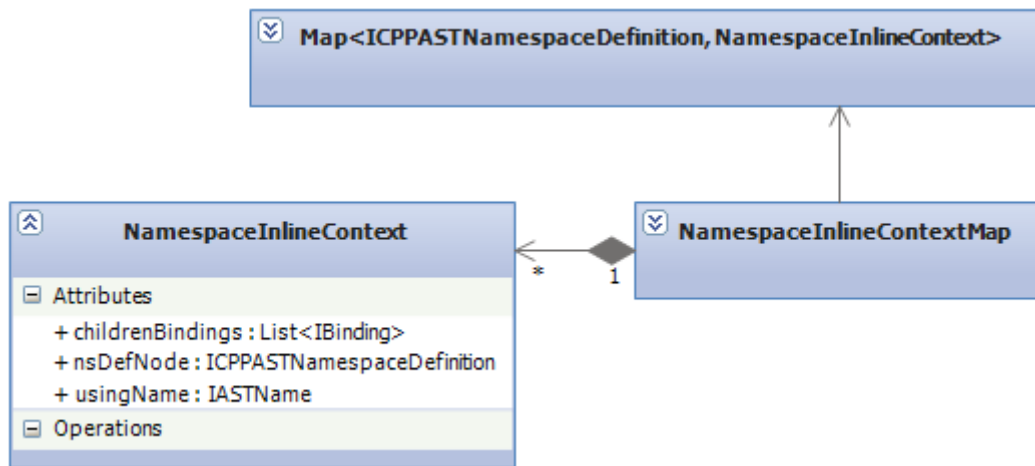


Figure D.2.: Class diagram of the `NamespaceInlineContext` class and companions used in the recursive search algorithm of the IUDIR refactoring implementation. The `NamespaceInlineContextMap` class is an adapter [GHJV95] for the generic type `Map<ICPPASTNamespaceDefinition, NamespaceInlineContext>`.

```
// file A.cpp
#include "B.cpp"
namespace A{ // merged with definition in B.cpp, same IBinding
    struct SA{ // collected name
        struct SSA{}; // not collected
    };
    namespace AA{} // collected name
}

int main(){
    using namespace A; // UDIR A, apply inline refactoring here
    SA sa(); // qualified with A
    SB sb(); // qualified with A
    SB2 sb2(); // qualified with B
    return 0;
}

// file B.cpp
namespace B{
    struct SB2{}; // collected name
}
namespace A{
    using namespace B; // UDIR B. starts recursive search of B
    struct SB{}; // collected name
}
```

Listing D.2: Foundation snippet for the explanation of the recursive search algorithm of the IUDIR implementation.

D.2. Open Issues

The indexer based IUDIR refactoring implementation has the following open issues.

D.2.1. Qualification of Implicit Operator Call

As illustrated in listing D.3, operator calls may also be affected by the IUDIR refactoring. Finding references of the operator definition in namespace OP is not a problem. Both calls inside the main method are found.

In the AST, these two calls are represented very differently, as illustrated in figure D.3. The explicit operator call (operator «(cout, s)) uses names to represent the operator. But the implicit call (cout « s) is represented using a binary expression. Using IASTN-selector.findNode(int offset, int length) for the reference node («) of the second call throws a NullPointerException.

namespactor currently ignores implicit operator calls and does not qualify them. In the listing below this results in a compilation error, because the argument dependent name lookup (ADL) does not work here, since no matching operator can be found in one of the namespaces of the argument types (A::ostream and A::string). The refactoring wizard shows a warning to notify the user about this circumstance, see figure D.4.

If the operator were defined in A, the code in listing D.3 would also work after only inlining the explicit operator call, because the ADL works. One solution to solve this

problem, might be to first transform the implicit operator call into an explicit operator call which than can be qualified.

```

namespace A{
    class string{
    public:
        string(char*){ }
    };
    class ostream{};
}

namespace OP{
    A::ostream& operator<<(A::ostream& o, const A::string&){}
}

int main(){
    using namespace A;
    using namespace OP; // start inline refactoring here
    string s("s...");
    ostream cout;
    operator <<(cout, s); // qualified as OP::operator <<(cout, s)
    cout << s; // operator is not a name => not qualified
    return 0;
}

```

Listing D.3: IUDIR refactoring affecting an explicit operator call. The implicit operator call is not inlined.

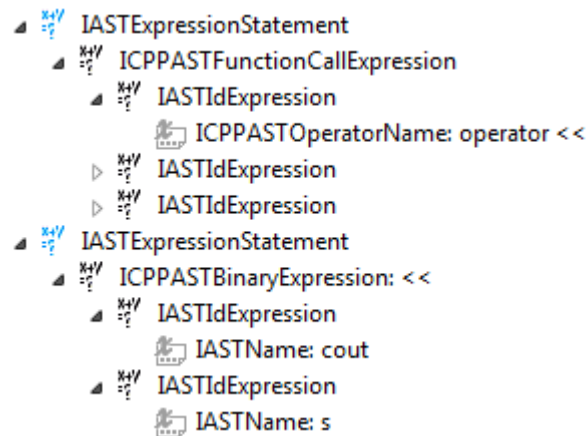


Figure D.3.: AST showing the difference between an implicit (top) and an explicit (bottom) operator call statement.

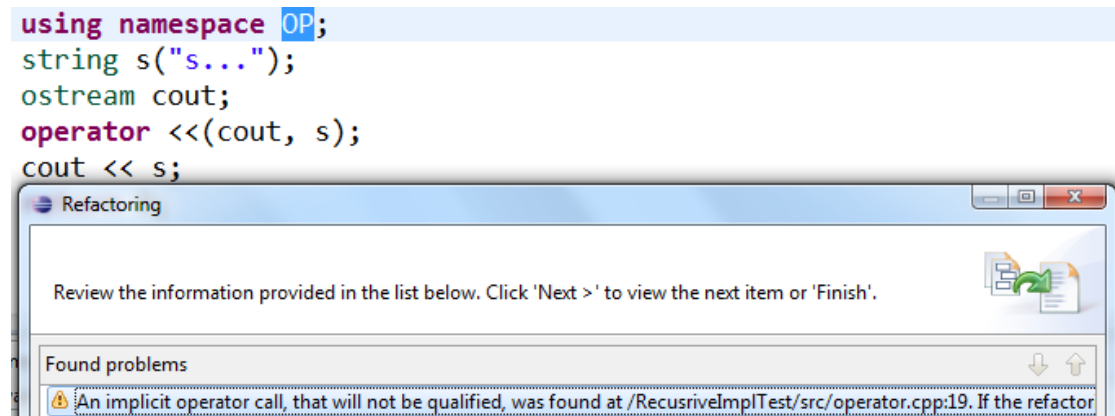


Figure D.4.: If an implicit operator call is affected by the IUDIR refactoring, the user is notified with a warning, because the implicit operator call will not be qualified.

E. User Manual

This user manual assumes that Eclipse CDT and namespaceator are already installed. The refactoring features of namespaceator work like the other refactorings Eclipse provides. Section E.1 introduces namespaceator by a simple example. In section E.2 all refactorings are presented. Section E.3 describes how the refactorings are invoked. Section E.4 presents the quick fixes and its usage.

E.1. Example of a Refactoring - Inline Using Directive

If the using directive in the listing E.1 is removed, the function call `doIt()` requires to be qualified. This work will be automated by the Inline Using Directive refactoring of namespaceator by selecting the using directive and invoking the refactoring in the Eclipse Refactoring Window Menu.

```
namespace Example{
    void doIt(){
        /* ... */
    }
}
int main(){
    using namespace Example;
    doIt();
}
```

Listing E.1: Example code for an Inline Using Directive refactoring.

After the refactoring was initiated, a window opens with a preview of the changes this refactoring would apply if the user confirms. The result of this refactoring is illustrated in listing E.2.

```
namespace Example{
    void doIt(){
        /* ... */
    }
}
int main(){
    Example::doIt();
}
```

Listing E.2: Result code of the foundation code in listing E.1 after the Inline Using Directive was applied.

E.2. Refactorings in namespactor

namespactor provides the following refactorings:

- Inline Using Directive: Removes a using directive and qualifies the affected names.
- Inline Using Declaration: Removes a using declaration and qualifies the occurrences of the affected name.
- Qualify an Unqualified Name: Fully qualifies an unqualified name with all required names.
- Extract Using Directive: Introduces a using directive for a qualified name and removes the name qualifier(s) of the affected qualified names.
- Extract Using Declaration: Introduces a using declaration for a qualified name and removes the name qualifiers(s) on the occurrences of the affected qualified name.

E.3. Run a Refactoring

This section describes how a refactoring is started. Based on a selection in the editor a refactoring can be started via the Eclipse refactoring window menu illustrated in figure E.1.

The menu entries of the refactorings largely describe to which refactorings it belongs. To start the inline using directive or inline using declaration refactoring the menu entry called "Inline Using ..." is used. This command decides itself which of the two inline refactorings is appropriate based on the selection done by the user.

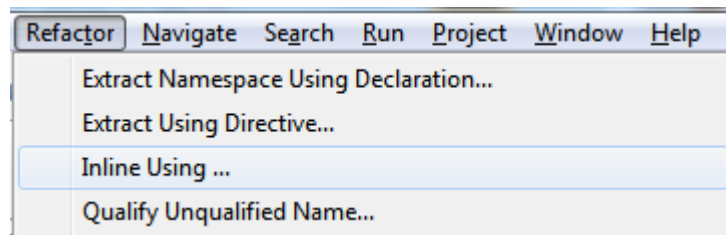


Figure E.1.: Eclipse Refactoring Window Menu with menu entries for the refactorings implemented in namespaceator.

The selection requires at least one character to be selected inside the using directive/declaration (inline refactoring) or the qualified name (extract refactorings). If more than the line of interest is selected, the last using directive/declaration (inline refactoring) or the last qualified name (extract refactorings and qualify an unqualified id refactoring) in the selection is the base for the refactoring. Figure E.2 illustrates possible selections, the upper two snippets with minimal selection and the lower two snippets with a wide selection. On the left are candidates for inline refactorings, on the right are candidates for the extract refactoring as well as for the qualify an unqualified name refactoring.

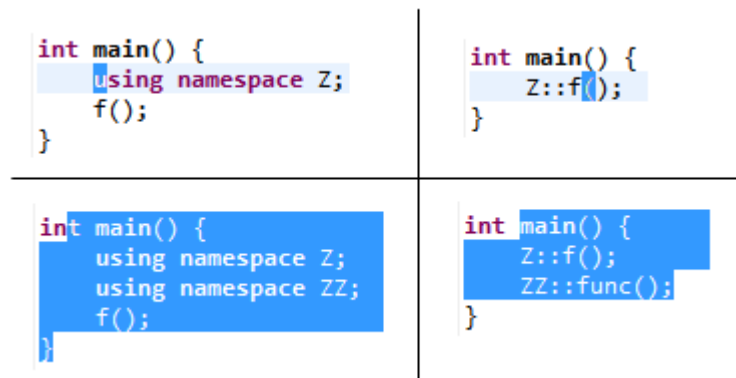


Figure E.2.: Selections in the editor of Eclipse.

E.4. Quick Fixes

namespaceator analyses the source code in a background process and reports potential problems. For each reported problem, namespaceator marks the problematic source code section and suggests to apply quick fixes. Figure E.3 illustrates a problematic source code section with a marker on the left and the proposed quick fixes to solve the problem in a tool tip.

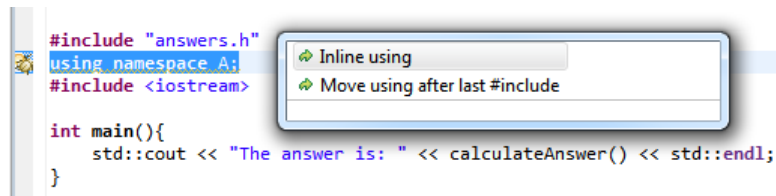


Figure E.3.: Example of bad source code.

namespactor provides following quick fixes:

- **Inline Using:** Invokes the inline using directive/declaration refactoring.
- **Move After Includes:** Moves a using directive/declaration after the last `#include` directive
- **Qualify Using Directive:** Invokes the qualify an unqualified name refactoring.

E.4.1. Problem Resolutions (Quick Fixes)

All of the reported problems can be solved by invoking one of the namespactor refactorings. Some problems have multiple resolutions and vice versa.

Inline Using (IU) Invokes the IU refactoring. Solves the problems: UDIRInHeader, UDECInHeader, UDIRBeforeInclude, UDECBeforeInclude

Move Using After Include (MoveAfterIncludes) Moves the problematic using statement after the last `#include` directive. Solves the problems: UDIRBeforeInclude, UDECBeforeInclude

Qualify Unqualified Name (QUN) Invokes the QUN refactoring. Solves the problems: QUNUDIR.

F. Project Management

This chapter provides an overview to project management related tasks such as project planning and spent time analysis.

F.1. Project Plan

The bachelor thesis lasts seventeen weeks from February 20. to June 15. 2012. The first version of the project plan F.1 was created in week one. During the project the initial project plan has experienced a few minor changes F.2.

We soon realised that the refactorings require more work than we originally planned. And that the work has not much to do with GUI but with a thorough analysis and challenging algorithms. The implementation of the inline and extract refactorings took double the time we estimated. But fortunately not on behalf of any other task. The GUI part was mainly provided by the LTK framework, and therefore only required one third of the estimated time.

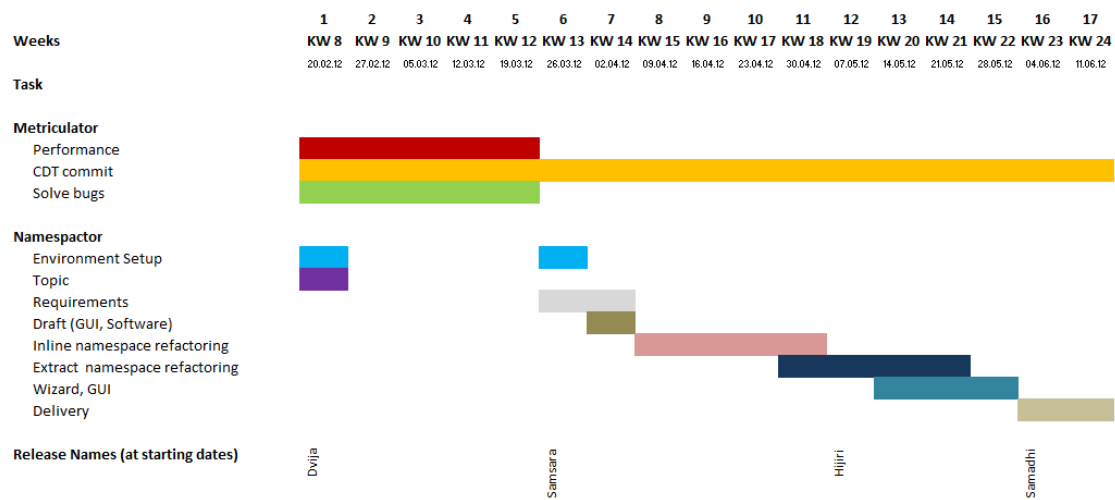


Figure F.1.: First version of the project plan.

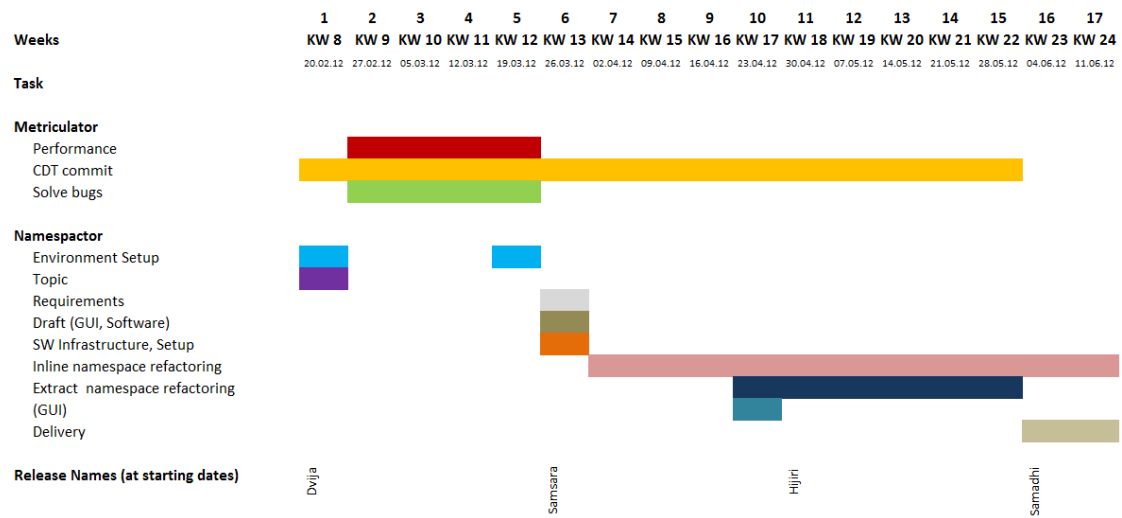


Figure F.2.: Latest version of the project plan.

F.2. Time Schedules

This chapter evaluates the time spent during the project. Figure F.3 shows the time spent per member per week. The bachelor thesis module is worth 12 ECTS. This means that the expected work per week of an average student to pass the module is about 21 hours¹. In average each of us worked about 447 hours in total, which is 87 hours (24%) above the expected 360 hours. In total, 895 hours of work were spent for both sub projects. We spent 695 hours for namespactor and 200 hours for metriculator.

¹12 ECTS * 30 hours per ECTS / 17 weeks

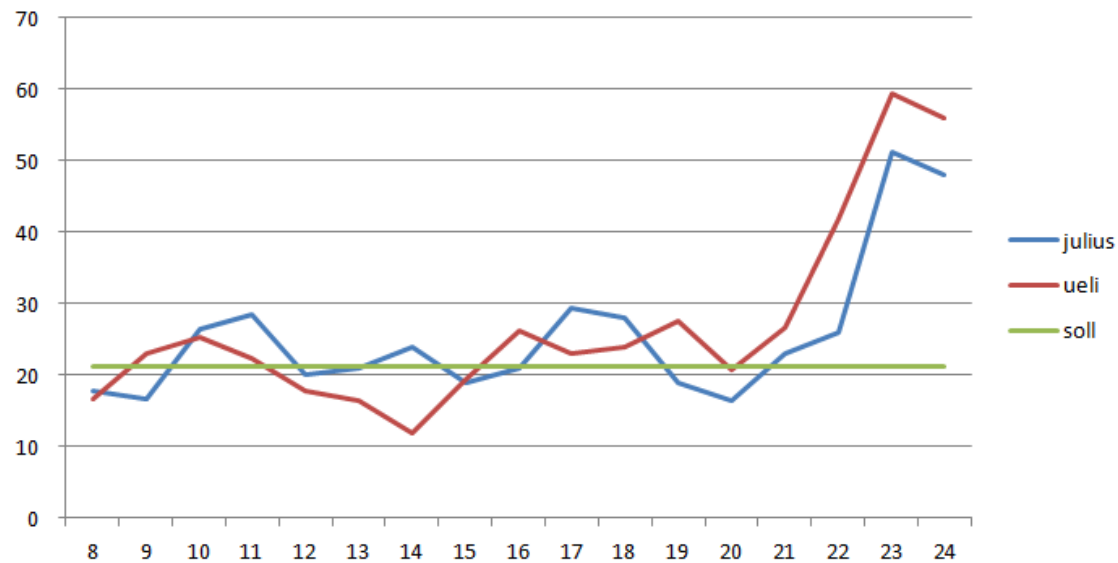


Figure F.3.: Time spent (in hours) per project member per week.

F.3. Personal Impression

From the very beginning of this thesis we both worked consequent and targeted enhancing the metriculator plug-in as well as to create a highly useful and simple to understand refactoring plug-in. In the following sub sections each team member writes about his personal impressions during this project. But first of all we would like to thank our supervisor Prof. Dr. Luc Bläser, for his valuable time and competent advices. Special thank goes to our advisor Thomas Corbat, who was always ready to generously assist us with technical problems as well as administrative issues. Further we would also like to thank Lukas Felber for his helpful advices in case of technical problems.

F.3.1. Ueli Kunz

The performance improvements we gained with metriculator are very satisfying. metriculator is now a reasonable metric tool for the CDT platform, this is also proved by the fact that in the first month metriculator has been installed over 40 times via Eclipse marketplace.

As we started working on metriculator I did not expect that the performance could be improved that much. I have learned, that it is, also with high level programming languages, very important to know what one is doing and that deliberately handling resources is important. It was also very interesting to see how the design has evolved step by step and that this enabled the implementation of an improved algorithm which lead to better performance.

I have never before done anything in the field of refactorings. I was curious and un-

certain about the subject at the same time. First I had to reboot my C++ knowledge. At the beginning it was a bit hard to see what a professional C++ developer would demand from a namespace refactoring tool, because I had insufficient experience to evaluate that and there was no other similar tool to learn from. But after some analysis and discussions with Thomas Corbat I was confident that we are on the right track.

The biggest obstacle was clearly the CDT framework. It has very poor documentation if any, since no literature exists and only sparse documentation is available online. The best sources accessible for us are inside the IFS, either from people working there or from other thesis documentations that were written in the past.

Handling C++ templates is something we undervalued. During the analysis we have not paid special attention to templates, for us template names were just another kind of names. But dealing with them, using the AST or the indexer, requires special treatment and some extra knowledge I did not have before.

Although it was kind of an experiment to divide a bachelor thesis into two projects for all involved parties, I am very glad that we had the chance to finalise the product we started during the semester thesis. This division required some organisational flexibility from all parties involved in the project. Everything went fine, but I would not recommend to let this become a habit. Especially the regulations and formalities given by the school are not designed for a forked thesis.

As already during the semester thesis, working with Julius Weder has been delightful and very enjoyable. I think we managed to create a fairly good refactoring tool in only eleven weeks.

F.3.2. Julius Weder

I think this bachelor thesis was a bit special because it consists of two projects. The first weeks were very interesting and also intensive. At first, I did not really had hopeful suspicions in significantly improving the performance of our semester thesis metriculator. But after some analysis and tests the speed up of the static analysis raised quite intense as well as my motivation.

It was again a great challenge to improve the design of metriculator and thereby discover how to improve the algorithm. Although it was the same project as in the semester thesis, the challenges were different. I learned a lot about memory allocation and the necessary releasing of the memory. It is even in Java a huge factor in developing a time critical software.

Although I had some experience with Eclipse plug-ins, CDT and the abstract syntax tree the refactorings in the second part of this thesis were a new, different and even more challenging work. At the beginning I had a lot of analysis to do, since my C++ skills and experiences still were not as advanced as it was necessary to create a useful namespace refactoring plug-in. This task took its time and was at a point a bit exhausting. Afterwards I think this was the key to create useful and for experienced C++ programmers meaningful refactoring features. We were able to deliver refactoring fea-

tures of good quality and good design even if big challenges and time consuming parts at this project were the algorithms to perform the refactorings.

Overall, I am proud of the work we have done and it was surely a great experience which further improved my professional skills and helps for future projects. At this point I would like to thank Ueli Kunz for being a very competent, helpful and pleasant project partner. I think this bachelor thesis proves that our teamwork is productive and equally enjoyable.

List of Figures

0.1. Example of bad source code.	v
0.2. Source code after the inline using directive refactoring was performed. The refactoring was initiated by a quick fix.	vi
0.3. Source code after the extract using directive refactoring was performed . .	vi
0.4. Memory allocation before and after the performance improvements analysing 1 mio. PSLOC.	vii
3.1. Layer diagram with the dependencies of namespactor.	33
3.2. Dependency graph of the namespactor core packages	35
3.3. Refactoring wizard showing the preview of an IUDIR refactoring.	36
3.4. Problem marker with suggested resolutions.	38
3.5. AST of template names	39
3.6. Scopes that play a role when comparing the indexer and AST lookup algorithms.	41
3.7. Sequence diagram of the IUDIR refactoring AST implementation.	47
3.8. Refactoring wizard shows a warning if the user tries to inline a using declaration that contains a generic template argument.	48
3.9. Sequence diagram illustrating the checkInitialConditions implementation of the extract refactorings.	55
3.10. Sequence diagram illustrating the visit implementation of the extract refactorings in the abstract EUNodeVisitor class.	56
3.11. Sequence diagram illustrating the buildReplacementName implementa- tion of the extract refactorings in the abstract EUNodeVisitor class. . .	57
3.12. Sequence diagram illustrating the collectModification implementation of the extract refactorings in the abstract EUNodeRefactoring class.	58
4.1. Before the performance improvements.	70
4.2. After the performance improvements.	71
4.3. Memory consumption before and after performance improvements.	72
4.4. Memory consumption before (left) and after (right) removing the AST bindings. The analysis on the right requires about one third of the memory in comparison.	74
4.5. Component diagram of metriculator and zest.cloudio.	77
4.6. AbstractNode composite hierarchy with AST related information stored in NodeInfo.	78
4.7. AbstractNodeInfo hierarchy with AST related information distributed in specialised NodeInfo classes.	79
4.8. Improved AbstractNode composite hierarchy.	81
4.9. AbstractNodeInfo hierarchy	82
4.10. AbstractNode hierarchy - the upper part of the hierarchy	82

4.11. AbstractNode hierarchy - the lower part of the hierarchy.	83
4.12. The jface TreeViewer and TableViewer components stretch the last column to fill the remaining horizontal space.	84
A.1. Eclipse test project using the CDTTesting plug-in	91
A.2. Concepts related to the CDTTesting plug-in and the RTS framework re- spectively.	92
A.3. Concepts of the RTS framework mapped to a RTS file.	93
A.4. Screenshot of the RTS editor and its outline view.	93
A.5. Screenshot of the DOM AST view.	95
D.1. Sequence diagram illustrating the IUDIR checkInitialConditions imple- mentation.	101
D.2. Class diagram of the NamespaceInlineContext class and companions. . . .	102
D.3. AST showing the difference between an implicit (top) and an explicit (bottom) operator call statement.	104
D.4. Implicit operator call warning.	105
E.1. Eclipse Refactoring Window Menu with menu entries for the refactorings implemented in namespactor.	108
E.2. Selections in the editor of Eclipse.	108
E.3. Example of bad source code.	109
F.1. First version of the project plan.	110
F.2. Latest version of the project plan.	111
F.3. Time spent (in hours) per project member per week.	112

List of Tables

4.1. Runtime performance measurement data before and after the improvements.	69
4.2. Class association mismatch between AbstractNode and AbstractNodeInfo hierarchy.	79

G. Nomenclature

AST Abstract Syntax Tree – An abstract representation of a program or source code, usually focusing on domain specific information.

VCS Version Control System – A software that helps managing multiple versions of files.

OSGi Specification for Java runtime service and modularisation platform.

PDE The Eclipse Plug-in Development Environment provides utilities to create, maintain, test and build Eclipse artefacts.

p2 Stands for provisioning platform and is the engine used to install plug-ins and manage dependencies in Eclipse.

IDE Integrated development environment is used to develop, compile and maintain source code written in a specific programming language

namespace Also called name scope, is a logical container that holds names of functions, classes, variables and other identifiers or symbols. A fully qualified name of an identifier thereby consists of the namespace(s) it is placed in and its own name.

Codan checker Codan uses checkers to analyse source code. Each checker is specialised in one problem, for instance unreachable code. Through extension points Codan allows third party tools to register other checkers. metriculator defines one checker per metric. As soon as a user invokes the Codan command on the UI, Codan automatically calls all registered checkers.

IUDIR Inline Using Directive Refactoring – One of the implemented refactorings in namespactor.

IUDEC Inline Using Declaration Refactoring – One of the implemented refactorings in namespactor.

EUDIR Extract Using Directive Refactoring – One of the implemented refactorings in namespactor.

EUDEC Extract Using Declaration Refactoring – One of the implemented refactorings in namespactor.

QUN Qualify an Unqualified Name – One of the implemented refactorings in namespactor.

Bibliography

- [Bec03] Kent Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003.
- [cdt] Announcement of metrculator in cdt-dev mailing list. <http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg23868.html>.
- [CDT11] Eclipse cdt project homepage. <http://eclipse.org/cdt/>, 2011.
- [cdt12a] Cdt bug 381031 - fully qualified names are not written by the namewriter. https://bugs.eclipse.org/bugs/show_bug.cgi?id=381032, 2012.
- [cdt12b] Cdt bug 382497 - pdomcppnamespace.getusingdirectives always returns empty array. https://bugs.eclipse.org/bugs/show_bug.cgi?id=382497, 2012.
- [cod11] Codan is a lightweight code analysis framework for the eclipse cdt platform. <http://wiki.eclipse.org/CDT/designs/StaticAnalysis>, 2011.
- [cpp11] Iso/iec 14882. PDF, 2011.
- [doo12] Doom3 source code. <https://github.com/TTimo/doom3.gpl>, 2012.
- [ecl11] jprofiler product page. profiling tool for the java virtual machine., 2011.
- [ecl12] Eclipse user interface guidelines. http://wiki.eclipse.org/User_Interface_Guidelines, 2012.
- [Fel12] Lukas Felber. Cdttesting framework. <https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting>, 2012.
- [FM05] Kerievsky Joshua Fowler Martin. Smells to refactorings. <http://www.industriallogic.com/papers/smellstorefactorings.pdf>, 2005.
- [Fow] Martin Fowler. *Refactoring : improving the design of existing code*. Addison-Wesley.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [ifs12] Institut für software. <http://www.ifs.hsr.ch/>, 2012.
- [JCo] Using jconsole to monitor applications. <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>.
- [jen12] Jenkins - ci server. <http://sinv-56013.edu.hsr.ch/jenkins>, 2012.
- [jfa12] Eclipse jface tableviewer. <http://help.eclipse.org/indigo/index.jsp?>

- topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjface%2Fviewers%2FTreeViewer.html, 2012.
- [llv11] Project homepage of the clang project. <http://llvm.org>, 2011.
- [ltk06] Eclipse. <http://www.eclipse.org/articles/Article-LTK/ltk.html>, 2006.
- [mav11] Maven download. <http://maven.apache.org/download.html>, 2011.
- [met11a] metriculator project home page. <http://tiny.cc/metriculator>, 2011.
- [met11b] metriculator report. <http://sinv-56013.edu.hsr.ch/redmine/attachments/1/metriculator.pdf>, 2011.
- [met12] metriculator - download the plugin-in at the eclipse marketplace. <http://marketplace.eclipse.org/content/metriculator>, 2012.
- [MI10] Roger Knöpfel Matthias Indermühle. Cdt c++ refactorings. <http://eprints3.hsr.ch/115/>, 2010.
- [mla08] *MLA style manual and guide to scholarly publishing*. Modern Language Association of America, New York, 2008.
- [nam] <http://sinv-56013.edu.hsr.ch/redmine/projects/namespacector/repository/revisions/master/entry/dev/ch.hsr.ifs.cdt.namespactor.rtstest/src/ch/hsr/ifs/cdt/namespacector/rtstest/testinfrastructure/JUnit4RtsRefactoringTest.java>.
- [nam12] namespacector project home page. <http://tiny.cc/namespacector>, 2012.
- [PDE11] Official help documentation for the eclipse plug-in development environment. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm, 2011.
- [red11] Redmine set up using mod passenger. http://www.redmine.org/projects/redmine/wiki/HowTo_Install_Redmine_in_Ubuntu#Ubuntu-1004-and-10041-using-Passenger, 2011.
- [Ref11] Refactoring explanation and definition. <http://www.ifs.hsr.ch/C-Refactoring.5821.0.html>, 2011.
- [Sch08] Doug Schaefer. <http://cdtdoug.blogspot.com/2008/11/code-analysis-and-refactoring-with-cdt.html>, 2008.
- [sou11] Sourcecloud plug-in for eclipse. <https://github.com/misto/Sourcecloud>, 2011.
- [Ste05] Devin Steffler. Dom ast view - cdt-dev mailing list announcement. <http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg04355.html>, 2005.
- [Str09] *Die C++ Programmiersprache*. 2009.
- [Sut] Herb Sutter. A modest proposal: Fixing adl (revision 2).
- [Sut00] Herb Sutter. Migrating to namespaces. 25(10):48, 50, 52, October 2000.

- [Sut02] Herb Sutter. *More exceptional C++ : 40 new engineering puzzles, programming problems, and solutions*. Addison-Wesley, Boston, 2002.
- [tur] Master's thesis.
- [vie12] Eclipse view contribution. http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_basicext_popupMenus.htm, 2012.
- [Vog11] Lars Vogel. Comprehensive tutorials for eclipse developers. <http://www.vogella.de>, 2011.
- [zes11] Zest, the eclipse visualization toolkit. <http://www.eclipse.org/gef/zest/>, 2011.

The versions of the documents, referenced to in this bibliography, that we used are stored in our VCS.