HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

semester thesis

One touch C++ code automation for Eclipse CDT

Toggle Function Definition

Martin Schwab, Thomas Kallenberg

December 22, 2010

Supervisor: Prof. Peter Sommerlad

During this semester thesis a code automation tool has been developed for the Eclipse C++ Development Toolkit (CDT) using the Eclipse refactoring mechanism. The resulting plug-in enables a C++ developer to move function definitions easily between header and source files.

The new plug-in differs from existing refactorings single keystroke interaction. The refactoring uses no wizard at all and is tolerant to imprecise code selection.

This document discusses the uses of the plug-in as well as the issues that had to be handled during the project. Students developing a new refactoring tool may have a look at the conclusion chapter, to not doing the same mistakes we did again and the Project Setup chapter to start with their own project quickly. Project setup hints are listed in the appendix.

Management Summary

In C++ there is the possibility to tell the compiler that there exists a function with a so called *declaration*. Since this does not specify what the function does, a *definition* is needed with the functionality.

```cpp
class A {
  int function(int param); //declaration
};

inline int function(int param)
{
  return param + 23; //definition
}
```

Listing 1: class with declaration and separated definition

Every definition is a declaration too.

```cpp
class A {
  int function(int param) { //decl. and def.
    return param + 23;
  }
};
```

Listing 2: class with declaration and definition

Since a declaration and definition can be separated, differences may occur between the signature of the definition and the signature of the declaration. This is not allowed. Now imagine that changing a function signature in C++ is an unthankful task. As an additional difficulty, a declaration and a definition may appear in two different files.

```cpp
#ifndef A_H_
#define A_H_

class A {
  void function();
};

#endif /* A_H_ */
```

Listing 3: Header file with a declaration

```cpp
#include "A.h"

void A::function() {
  return
}
```

Listing 4: Source file containing definition

If a signature changes, these changes must be made in two files: the header file and the implementation file. More than once programmers forget to change the signature in one place which results in compile errors and unnecessary time consuming error correction.

Toggle Function Definition

Refactorings are solving such problems by automating dependent changes to source code, so less errors are introduced by hand.

Toggle Function Definition moves a function definition inside an Eclipse CDT source editor from one position to another and preserves correctness.

This is done by searching for a function definition or declaration next to the user's code selection. Then, according to the found element, it's sibling is searched. After that, the signature of the definition is copied and adapted to the new position. The new definition gets inserted and the old definition is removed. If no separate declaration existed before, the old definition is replaced by a newly created declaration.

All this is done without any wizard and kept speedy to not break the work flow. The refactoring is bound to the key combination Alt-Shift-V.

```cpp
class A {
  int function(int param) {
    return 42; | //<- cursor position
  }
};
```

Listing 5: Startposition of toggling

```cpp
class A {
  int function(int param) {
    return 42;
  }
};

X // <- new position is here
```

Listing 6: New position is found

```cpp
class A {
  int function(int param);
};

inline int function(int param) {
  return 42;
}
```

Listing 7: Class with declaration and inlined definition

Toggling again moves the definition out of the header file to the implementation file that should contain the actual functionality. If `function(int param)` from listing 7 is toggled, the definition will end up in the implementation file as shown in listing 8.

```cpp
#include "A.h"

int A::function(int param) {
  return 42;
}
```

Listing 8: Defintion in an implementation file

Quick Implement Function

To obtain some coding flow and due to the fact that *Implement Method* does not really fit into this fast-toggle working style, this semester thesis re-implemented the implement function as a *Quick Implement Function*.

The idea behind this is the following: The developer starts writing a class. He comes to the point where he has written the first declaration and then, by using a keystroke Alt-Shift-Z, the declaration is replaced with a definition containing an appropriate return statement.

```
class A {
  int function();|//<-cursor position
};
```

Listing 9: Situation before quick implement

```
class A {
  int function()
  {
    return int(); //new generated definition
  }
};
```

Listing 10: Situation after quick implement

The developer can continue to write the functionality of `function()` and toggles the definition with the other keystroke Alt-Shift-V to the header file.

Thanks

We would not have been able to achieve this project without the help of others. A big thanks goes to all of these people.
Specially we would like to thank Prof. Peter Sommerlad for the original idea of the toggle refactoring, for supervising the project and for various cool ideas in many problems we encountered, Lukas Felber who provided us with instant solutions where we struggled to continue, Emanuel Graf for his ideas and explanation in every subtopic of the CDT project and Thomas Corbat for help and ideas for various subtopics like comment handling.
Another big thank goes to our families and friends who were missed out a little bit during our semester thesis.

Contents

1 Introduction

1.1 Current Situation

The Eclipse Java Development Toolkit (JDT) has a large set of both quick and useful refactorings. Its sibling the C++ Development Toolkit (CDT) offers just a small range of such code helpers today. In addition, some of them don't work satisfactory: Currently, extracting the body of a hello world function takes more than three seconds on our machines. Reliability? Try to *extract constant* the hello world string of the same program. At the time of this writing, this still failed.

Bachelor students at HSR may visit a C++ programming course where Eclipse is used to solve exercises. For the authors of this document, it was clear after a while that touching the refactoring buttons was a dangerous action because in some cases described above they broke your code. Compile errors all over the place and difficult exercise assignments didn't make our life easier.

1.2 Motivation

One annoying problem in C++ is the separation of the source and the header files. This is a pain point for every programmer. Forgetting to update the function signature in one of the files will result in a compilation error causing either lack of understanding for beginners or loss of time.

After two minutes of compile error hunting because of a function signature that was out of sync in the header and the implementation file, it may be asked: Why has nobody yet implemented a solution to prevent such an error?

Refactorings in CDT have a big field of such interesting and unresolved problems. With the support of the Institute of Software IFS at HSR Rapperswil, these problems could be solved. The Institute for Software with its group around Professor Peter Sommerlad and Emanuel Graf has been working on Eclipse refactorings for a long time. Since 2006 nine Eclipse refactoring projects have been completed.

1.3 What has been Planned

During this semester thesis it was planned to introduce and improve one or more refactorings to the Eclipse CDT project. They will now be introduced by priority.

1.3.1 Toggle Function Definition

The goal of this refactoring is to automate the process of moving the definition of a function from one place to another as shown in Listings 1.1 and 1.2:

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

class ClearClass {
  void bigfunction() {
    /* implementation */
  }
};

#endif
```

Listing 1.1: Initial situation: member function defined inside a class

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

class ClearClass {
  void bigfunction();
};

inline void ClearClass::bigfunction {
  /* implementation */
}
```

```

#endif
```

Listing 1.2: Separated definition after toggling

The example shows that moving the function body does not only involve copying code but also adding a scope operator :: to the new definition. In addition, the former definition had to be replaced by a plain declaration. There are several such changes required depending on the scope and properties (static, virtual, etc.) of the function. Toggling multiple times should bring the code back to the original position in almost no time.

Throughout this documentation, this functionality is referred to as being a *refactoring* as it uses the refactoring facility of the Eclipse LTK [Fre06]. However, *Toggle Function Definition* actually is a code generator. The idea for this ”refactoring” was introduced by project advisor Prof. Peter Sommerlad.

Goal of this thesis is to realize this idea in form of an Eclipse plug-in and to make it fast enough to become a good alternative to editing the functions manually.

1.3.2 Implement Member Function

The current CDT plug-in already includes an *Implement Member Function*. However, it is slow and does not fit together with the newly created *Toggle Function Definition*. It breaks the coding flow for adding functionality to classes with unneccessary wizards, which could be a reason not to use the new toggle functionality subsequently. This code generator shall support *Toggle Function Definition* by providing a quick way to create a new function given an existing declaration.

Depending on the success of the implementation with the first refactoring, it is planned to re-implement the *Implement Function* refactoring.

1.3.3 Override Virtual Member Function

This code genrator may share some functionality with the above ones. Its goal is to help the user override multiple member functions of an inherited class.

1.4 Objectives

These are the basic aims for the project:

- Toggling between `in-class`, `in-header`, `separate-file` and back again to `in-class` works for basic and some frequent special cases.
- Project organization: Fixed one-week iterations are used. Redmine is used for planning and tracking time, issue tracking and as information radiator for the supervisor. A project documentation is written. Organization and results are reviewed weekly together with the supervisor.
- Quality: Common cases are covered with test cases for each refactoring subtype.
- Integration and Automation: Sitting in front of a fresh Eclipse CDT installation a first semester student can install our refactoring using an update site as long as the functionality is not integrated into the main CDT plug-in.
- To minimize the integration overhead with CDT it will be worked closely with Emanuel Graf as he is a CDT commiter.
- At the end the project will be handed to the supervisor with two CD's and two paper versions of the documentation. An update site is created where the functionality can be added to Eclipse. A website describes in short words the functionality and the project vision.

1.4.1 Advanced Objectives

All basic goals will be achieved. Additionally:

- Toggling function is fast. Less than a second.

Re-Implement the "Implement Function" feature.

- A new function block is created with nearly no delay right below the function signature.

- A default return statement is created when the block is created.
- If the return statement cannot be determined, a comment is inserted into the block.

1.4.2 Further Objectives and Outlook

If there is enough time, an *Override Virtual Function* is implemented. Additionally, content assist may be implemented. This could be part of a bachelor thesis which continues and completes the work done in this semester thesis.

1.5 Expected Outcome

Implement member function and the toggle key are written to work in synergy. First write the declaration for a (member) function in the header or class definition, then a hot-key is used to implement the function. At this point the toggle key may be hit at any time to move the function to the appropriate position and continue with the next new member function.

1.6 Project Duration

The semester thesis starts on September 20th and has to be finished until December 23rd, 2010.

2 Specification

This section describes how the different code automation mechanisms have been analyzed and designed.

2.1 Toggle Function Definition

Good code should separate interface and implementation. However, it is annoying to copy function signatures from the header file to the implementation file or vice versa. This process shall be automated.

Toggle Function Definition moves the code of a member function between different possible places, preserving the declaration statement inside the header file. What the different places are, in which direction the code may be moved and in which situation the refactoring may be invoked is described in the following chapters.

2.1.1 Activation

For the selected function, a function definition must exist in the current or an associated file. If no definition exists, the refactoring aborts. There may be more than one declaration. However it is not specified to which one will be toggled. It will be toggled to any declaration that is found first.

The refactoring shall allow selections anywhere inside the function, whether inside the signature, function (try) body, a catch handler or a template declaration.

If functions are defined inside a function body which is non-standard but allowed by some compilers, the outermost function parent should be toggled. [Fre10]

2.1.2 Three Positions for Function Definitions

In C++ there are three possible positions where a function definition may occur. Listing 2.1 shows an example where the definition of a member function is placed inside its class definition. New code blocks created by *Implement Member Function* are placed inside the class definition too. Placing implementation code right in the class definition is also the most intuitive behaviour for Java developers.

```
#ifndef A_H_
#define A_H_

namespace N {
class A {
  int function() {
    return 0;
  }
};
}

#endif /* A_H_ */
```

Listing 2.1: In-class implementation in A.h

To keep the interface clear, function definitions may be placed outside the class definition but are still located in the same (header) file. Such a function is called *inlined*. See Listing 2.2.

For templates, this is the only position outside the class definition where the implementation may be placed, due to problems of the `export` keyword [SP03]. This means: templated functions cannot be placed outside the header file. Except for functions which are specially marked with above `export` keyword.

Listing 2.2 shows an example of what will be called *in-header situation* throughout this document.

```
#ifndef A_H_
#define A_H_

namespace N {
class A {
  int function();
};

inline int A::function() {
  return 0;
}
}

#endif /* A_H_ */
```

Listing 2.2: In-header implementation in A.h

To separate the implementation from the interface more clearly, a separate source file may be used for the definitions while the declarations remain in the header file.

An example for this position of a function definition is shown in listings 2.3 and 2.4. This position will be called *in-implementation* throughout this document.

```
#ifndef A_H_
#define A_H_

namespace N {
class A {
  int function();
};
}

#endif /* _A_H */
```

Listing 2.3: A.h, with declaration

```
#include "A.h"

namespace N {
int A::function() {
  return 0;
}
}


.
```

Listing 2.4: A.cpp, with definition

2.1.3 Basic Scenarios

Depending on the current selection, a different strategy needs to be applied to move the function definition. All supported toggling situations and their special cases are listed in this section.

Free Functions (Non-Member Functions)

Functions which are not member of a class are so called *non-member functions*. In this document they are called *free functions* to distinguish them more clearly.

Toggling for plain free functions shall be possible at two positions:

1. Toggle from in-header to in-file

2. Toggle from in-file to in-header

Example

Let us assume a free function definition in a header file with no further declaration specified is toggled. Listing 2.5 shows the situation before toggling.

```cpp
int freefunction() {
  return 42;
}

int main() {
  return 0;
}
```

Listing 2.5: A.cpp, initial situation

First it is checked if there exists a file with the same name as the original implementation file. A.h in this example. If not, a new file is created with the appropriate include guards. See listing 2.6.

```cpp
#ifndef A_H_
#define A_H_

#endif /* A_H_ */
```

Listing 2.6: Newly created A.h

Subsequently, the `freefunction()` definition is moved into the header file as shown in listing 2.7

```cpp
#ifndef A_H_
#define A_H_

int freefunction() {
  return 42;
}

#endif /* A_H_ */
```

Listing 2.7: Inserted `freefunction()` in A.h

If toggled again, the declaration of `freefunction()` remains inside the header file, while the definition is inserted into the implementation file and an include statement is inserted at the beginning if of the implementation file. Listing 2.8 and 2.9 shows the end situation.

```
#ifndef A_H_
#define A_H_

int freefunction();

#endif /* A_H_ */


.
```

Listing 2.8: A.h, inserted declaration

```
#include "A.h"

int freefunction() {
  return 42;
}

int main() {
  return 0;
}
```

Listing 2.9: A.cpp, inserted definition

If `freefunction()` is toggled again, the declaration in the header file has to be replaced by the definition which is removed from the implementation file, resulting in a header file already shown in listing 2.7.

Member Functions

For functions inside classes, toggling is expected to be available for three positions:

1. Toggle from in-class (to in-header)
2. Toggle from in-header (to in-file)
3. Toggle from in-file (to in-class)

Example

The starting point for toggling member functions could be a class with a function definition inside like in listing 2.10.

```
#ifndef A_H_
#define A_H_

```

```
namespace N {
class A {
  virtual void function() {
    return;
  }
};
}
#endif /* A_H_ */
```

Listing 2.10: A.h, function definition inside class declaration

Function `function()` needs to be toggled. The next position of the definition is ouside of the class but kept in the namespace definition. The definition is replaced by a declaration as in 2.11.

If there is no namespace definition, the function definition will be placed below the class in the header file. See listing 2.12.

If there are any special keywords like `virtual` or `static`, these are adapted to the new definition. Definitions in the header file need the prefixed keyword `inline`. The keyword `virtual` is only allowed inside a class definition.

```
#ifndef A_H_
#define A_H_

namespace N {
class A {
  vitual void function();
};

inline void A::function() {
  return;
}
}
#endif /* A_H_ */
```

Listing 2.11: A.h, function definition outside of class definition in header

```
#ifndef A_H_
#define A_H_

class A {
  virtual void function();
};

inline void A::function() {
  return;
}

#endif /* A_H_ */
```

Listing 2.12: A.h, function definition outside class definition without namespace

If `function()` gets toggled again, the definition is moved to the implementation file and if necessary a namespace definition is created where the function gets inserted. Nothing remains outside of the class definition in the header file and the declaration in the class does not change.

```
#ifndef A_H_
#define A_H_

namespace N {
class A {
  void function();
};
}

#endif /* A_H_ */
```

Listing 2.13: A.h, after moved definition

```
#include "A.h"

namespace N {
void A::function() {
  return
}
}


.
```

Listing 2.14: A.cpp with definition

If `function()` is toggled once again, the original starting position from listing 2.10 is reached.

2.1.4 Special Cases

Not every function may be toggled between the three positions and some cases require additional work before they may be toggled. Those special case are listed in this section.

Namespaces

If the moved function definition is contained inside a namespace definition, the function definition is moved with regard to the namespace. This means when toggling from *in-class* to *in-header* the definition is inserted before the namespace is closed in the header file.

```
namespace N { //namespace found
class A {
  void function() {
    return;
  }
};
X // <- new position inside namespace
}
```

Listing 2.15: A.h

If the function is toggled from *in-header* to *in-implementation* and there is no namespace definition, a new namespace is created.

```
#include "A.h"

namespace N {
//namespace created
}

#endif /* _A_H */
```

Listing 2.16: A.cpp, new namespace created

```
#include "A.h"

namespace N {
int A::function() {
  return 0;
}
}
```

Listing 2.17: A.cpp, insterted function

Namespace definitions that become empty after removing the last function definition shall be deleted.

```cpp
#include "A.h"

namespace N {
//empty namepsace
}

#endif /* _A_H */
```

Listing 2.18: A.cpp, empty namespace

```cpp
#include "A.h"


//no namespace


.
```

Listing 2.19: A.cpp, removed empty namepsace

Templated Member Functions

Another exception is a templated member function that may only be toggled inside the same header file. There, two strategies are interesting.

1. Toggle from in-class to in-header
2. Toggle from in-header to in-class

Example

The starting situation is shown in listing 2.20.

```cpp
#ifndef A_H_
#define A_H_

template <typename T>
class A {
  void function(T & t) {
    return;
  }
};

#endif /* A_H_ */
```

Listing 2.20: A.h, *in-class* definition with template parameters

Toggling `function(T & t)` now does not differ from toggling a non templated member function and will result in listing 2.21.

```cpp
#ifndef A_H_
```

```
#define A_H_

template <typename T>
class A {
  void function(T & t);
};

template <typename T>
inline void A::function(T & t) {
  return;
}

#endif /* A_H_ */
```

Listing 2.21: A.h, *in-header* definition with template parameters

Toggling again will put the definition back to its original position in the class definition as shown in listing 2.20. When a template definition is in another file than the declaration, the export keyword is needed. However, this is not supported by many compilers.

2.1.5 Expected Result

Toggling Member functions should work for the default cases. "Normal" member functions should be toggled fast without producing inconsistent code.

Templated member functions should be supported in a normal way. It is not the idea to give support for obscure tricks with template metaprogramming or other strange things which the usual C++ programmer does not use.

Additionally free (non-member) functions should be supported too. This means the refactoring should work for C projects in Eclipse as good as for C++ projects.

2.2 Quick Implement Function

Goal of this functionality is to offer an efficient way to append a minimal function body to an existing function declaration.

2.2.1 Activation

This refactoring shall be active as soon as a function declaration is selected that has no associated definition. The original idea was to use this refactoring on declarations without a trailing semicolon. This is problematic because the state without a semicolon is saved before the refactoring starts. This however results in a so called *problem node*. This means the parser of the compiler found an error in this source code range, resulting in a corrupt index.

In the short time left to implement this feature it was not achieved to parse the problem node and generate correct code out of it.

So, an already completed function declaration can be transformed to a function definition by using the "Quick Implement Member Function" hot-key which creates a body with default empty return statement. If the return statement can not be created, e.g. the return type of the function is a reference, no return statement is created in the body.

2.2.2 Example

```
#ifndef A_H_
#define A_H_

class A {
  int function();
};

#endif /* A_H_ */
```

Listing 2.22: A.h, with declaration and no definition

Selecting `function()` and using the *Quick Implement Function* key Ctrl-Shift-Z results in the following listing 2.23.

```
#ifndef A_H_
#define A_H_

class A {
  int function()
  {
    return int()
  }
};

#endif /* A_H_ */
```

Listing 2.23: A.h, with declaration and no definition

2.2.3 Expected Result

As described above, functions may only be toggled when they provide a function body. This refactoring shall provide a facility to create an function body with a default return value to enable *Toggle Function Definition*.

The re-implementation of *Implement Function* must be very fast.

2.3 Override Virtual Member Function

No deeper investigation has been done for this refactoring since it was not implemented during the project.

3 Implementation and Solution

From the three specified refactorings, *Toggle Function Definition* has been implemented in depth. This chapter explains how the refactoring was implemented and how it was tested.

3.1 Implementation approach

At the beginning, as many different cases as possible were collected on the project wiki to gain a view on what had to be realized, what was planned to take into scope and what had nothing to do with toggling function definitions. Some cases were simple, some exotic. The simplest one, toggling from inside a class to the same file outside the class, was chosen to be implemented first (See listings 1.1 and 1.2).

Before, a skeleton plug-in was built with a `NullRefactoring` to try whether it was possible to develop a separate plug-in instead of directly manipulating the CDT source code. By this approach it was assured that the developed plug-in may be deployed easily even without being integrated into CDT.

After the first refactoring was implemented, more cases were added by order how a member function is toggled circularly. Mostly it was worked with a test driven development approach. First write a test and then implement the functionality to get a positive test restult.

3.2 Architecture

In Eclipse, most of the architecture of a plug-in is already given. Some specialties of the toggle refactoring implementation are presented in this section.

3.2.1 Class Diagram

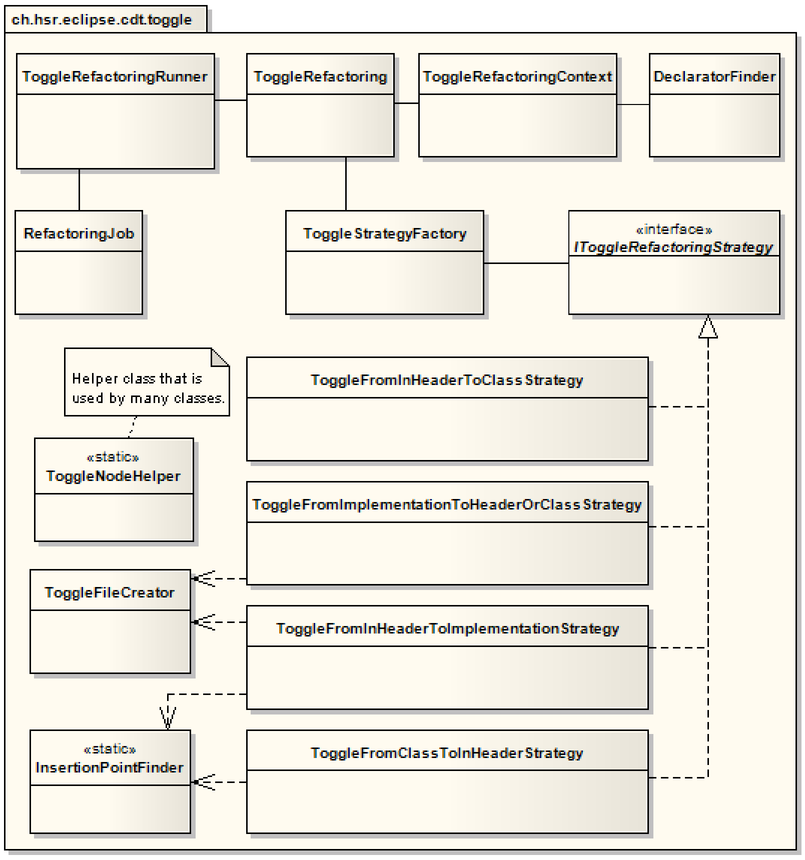

ch.hsr.eclipse.cdt.toggle
ToggleRefactoringRunner
ToggleRefactoring
ToggleRefactoringContext
DeclaratorFinder
RefactoringJob
ToggleStrategyFactory
«interface»
IToggleRefactoringStrategy
Helper class that is used by many classes.
«static»
ToggleNodeHelper
ToggleFromInHeaderToClassStrategy
ToggleFromImplementationToHeaderOrClassStrategy
ToggleFileCreator
ToggleFromInHeaderToImplementationStrategy
«static»
InsertionPointFinder
ToggleFromClassToInHeaderStrategy

Figure 3.1: class diagram of Toggle Function Definition

3.2.2 Basic Call Flow

The sequence diagram in figure 3.2 illustrates the basic call flow when *Toggle Function* is invoked.

3.2.3 Strategies

The way to toggle from one place to another differs depending on the current position. Having all logic in the same unit would need a complex conditional structure which is on one side confusing and on the other side slow.

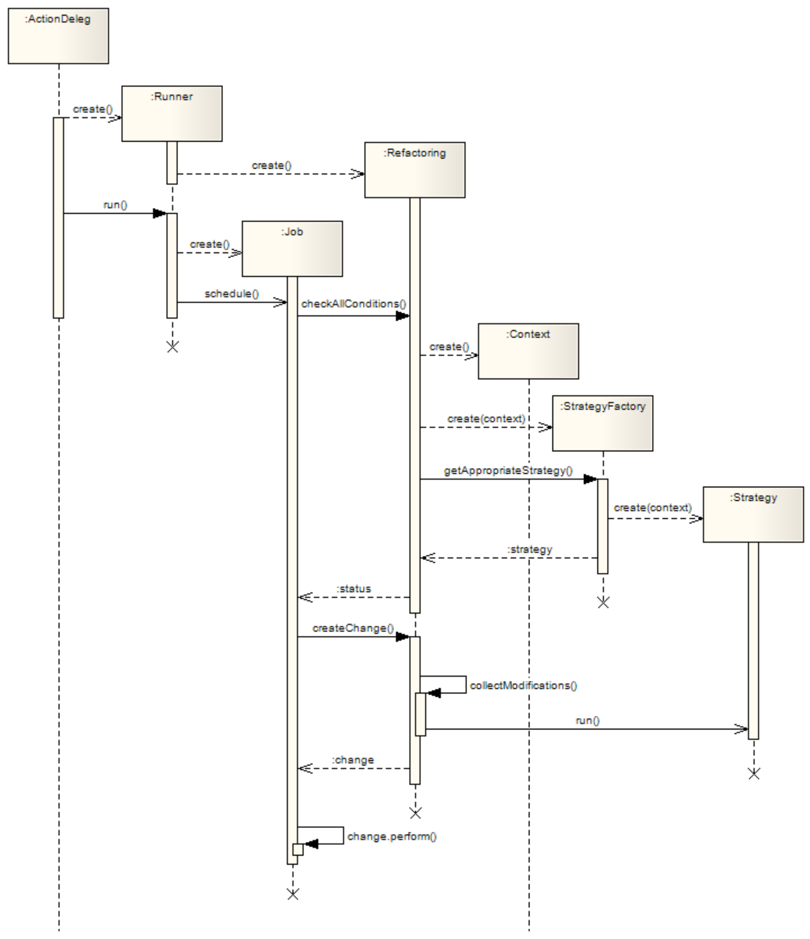


Figure 3.2: Basic call flow when toggling a function definition

Consequently, a strategy pattern based code structure was introduced. For toggling a simple not templated member function, three strategies were used. With the help of these, member functions may be toggled circularly.

- ToggleFromClassToInHeaderStrategy
- ToggleFromInHeaderToImplementationStrategy
- ToggleFromImplementationHeaderOrClassStrategy

To support templated classes, another strategy is required which toggles from *in-header* back to *in-class* as explained in section 2.1.4.

This strategy is specially implemented to support templated functions.

- ToggleFromInHeaderToClassStrategy

All these strategies implement an interface with a `run()` method taking a `ModificationCollector` argument to collect the changes to be applied to the source code.

```
public interface IToggleRefactoringStrategy {
  public void run(ModificationCollector col);
}
```

Listing 3.1: IToggleRefactoringStrategy

An interface was chosen because an abstract class containing all the methods needed by the various strategies was too big and unclear. This was solved with an interface and a static helper class named `ToggleNodeHelper`.

3.2.4 ToggleNodeHelper

`ToggleNodeHelper` contains a lot of methods which could be reused by other projects. It inherits from NodeHelper to make the integration of these methods as smooth as possible.

3.2.5 Context

The `ToggleRefactoringContext` is used to collect and store information about definitions, declarations and their corresponding translation units.

The context is then passed to the strategy factory. See section 3.2.6. Then the factory creates the strategy and passes the context to this specific strategy. The strategy retrieves all the needed information about the current situation from the context.

The context was introduced to prevent the code smell *Long Parameter List* [Fow99]. A common refactoring for this smell is to introduce a *Parameter Object* which consolidates all arguments.

The context searches the information by its own due to the fact that context would just be a very small data class and yet another class would be needed to search and collect the information, builds and returns the context.

3.2.6 Strategy Factory

The *ToggleStrategyFactory* is used to decide which strategy should be considered based on the passed context. The strategy makes various checks and decides which strategy will be returned.

```
public IToggleRefactoringStrategy getStategy() {
  if (context.getDefinition() == null) {
    throw new NotSupportedException(...);
  }
  ...
  if (isInClassSituation()) {
    return new ClassToInHeaderStrategy(context);
  }
  if (isTemplateSituation()) {
    return new HeaderToClassStrategy(context);
  }
  ...
}
```

Listing 3.2: IToggleRefactoringStrategy

3.2.7 Stopping with Exception

Refactorings that use a wizard may communicate with the user by displaying warnings and errors. Those are internally collected in a `RefactoringStatus` object by the refactorings.

This approach was used too until it became too tedious to always pass and process the status parameter in all classes used during `ToggleRefactoring`'s `checkInitialConditions(...)`. Every method that needed to abort the checking process needed to use code as in listing 3.3.

```
public RefactoringStatus findSomeNode() {
  ...
  if (hadSomeProblem) {
    initStatus.addFatalError("fatal");
    return initStatus;
  }
  ...
  return initStatus
}
```

Listing 3.3: Exemplary use of the RefactoringStatus

The latter use of `RefactoringStatus` consumes five lines of code and uses up the return value in each method. To solve this, a `NotSupportedException` was introduced which may be thrown by any client of `checkInitialConditions`. There, the exception is catched and transformed into a RefactoringStatus as shown in 3.4

```java
public RefactoringStatus checkInitialConditions(..
  try {
    ...
  } catch (NotSupportedException e) {
    initStatus.addFatalError(e.getMessage());
  }
  return initStatus;
}
```

Listing 3.4: `checkInitialConditions` forwarding an exception

3.2.8 Implications of not Using a Refactoring Wizard

No wizard was used for this refactoring since it must be fast and may be executed several times in succession. When using a wizard, the *RefactoringWizardOpenOperation* handles the execution of the refactoring inside a separate job. Since the toggle refactoring does not use the wizard, a separate job had to be scheduled by the ActionDelegate.

In addition, the undo functionality had to be implemented separately. When the changes are performed, they also return the undo changes that are needed by the UndoManager. The functionality of the `ToggleRefactoringRunner` is described in the following section.

3.2.9 Running the Refactoring

Present refactorings use a `RefactoringWizard` together with a `WizardOpenOperation` to execute a refactoring. Listing 3.5 shows CDT's `HideMethodRefactoringRunner` run method as an example.

```java
public void run() {
  CRefactoring refactoring =
     new HideMethodRefactoring(...);
  HideMethodRefactoringWizard wizard =
     new HideMethodRefactoringWizard(refactoring);
  RefactoringWizardOpenOperation operator =
     new RefactoringWizardOpenOperation(wizard);
  operator.run(shellProvider.getShell(),
     refactoring.getName());
}
```

Listing 3.5: Shorted run method of HideMethodRefactoringRunner

As discussed before, no wizard is used to start *Toggle Function Definition*. Instead, the refactoring is executed directly by the `ToggleRefactoringRunner`. This means that the latter class needs to take care of what the `WizardOpenOperation` was responsible before.

The responsibilities of `ToggleRefactoringRunner` and `RefactoringJob` are explained in the following sections.

Run a Separate Job

Why does the refactoring have to run in a separate job?

Toggle Function Definition does not use wizards and therefore has no UI blocking modal dialogs. Any process like waiting for the indexer would just freeze the user interface. Running the refactoring in a separate job allows the user to continue using Eclipse as long as he does not change the affected source code.

Running the refactoring in a separate job is straightforward:

```java
public void run() {
  ...
  new RefactoringJob(refactoring).schedule();
}
```

Listing 3.6: ToggleRefactoringRunner starting the job

Avoid Concurrent Refactoring Instances

Why is queuing refactoring jobs not allowed?

As an addition consequence of not using modal dialogs, it is possible to invoke another concurrent instance of the refactoring. Even though the refactoring runs in a separate job, it was decided to not allow multiple instances of the refactoring. Note that selected code could be removed during refactoring and a subsequent refactoring's selection would be invalid.

To decide whether another refactoring is still running, the `RefactoringJob` is assigned to a special kind of jobs by overriding the `belongsTo()` method of `org.eclipse.core.runtime.jobs.Job`. See listing 3.7.

```
public final static Object FAMILY_TOGGLE_DEFINITION
    = new Object();

@Override
public boolean belongsTo(Object family) {
  return family == FAMILY_TOGGLE_DEFINITION;
}
```

Listing 3.7: RefactoringJob is assigned to a separate family of jobs

With the help of the overriden `belongsTo()` method, the job manager can now check whether another job of the same family is running. Listing 3.8 shows how a second refactoring instance is avoided by the *ToggleRefactoringRunner*.

```
public void run() {
  Job[] jobs = Job.getJobManager()
   .find(RefactoringJob.FAMILY_TOGGLE_DEFINITION);
  if (jobs.length > 0) {
    CUIPlugin.log(...);
    return;
  }
  new RefactoringJob(refactoring).schedule();
}
```

Listing 3.8: ToggleRefactoringRunner avoiding a second refactoring instance

Execute Refactoring and Support Undoing it

Running a refactoring is essentially calling its methods `checkAllConditions` and `createChanges`. The returned changes are performed using `changes.perform(...)`. The `perform` method returns the changes needed to undo the performed changes.

The process of registering the changes at the undo manager is long and may be looked up in the `RefactoringJob` class.

3.3 Testing and Performance Environment

This section introduces some approaches to simplify testing and monitoring of performance.

3.3.1 Normal Testing

The test coverage for the toggle refactoring reached over 80%. Mainly the refactoring tests from CDT were used for default testing. The test files were divided into the various C++ features. They may require special handling or they must be supported because they are simple default cases. These files have the inner structure in which the toggle order was implemented. Namely from *in-class* to *in-header*, from *in-header* to *in-implementation* and from *in-implementation* to *in-class*.

After a problem was found, an issue was created in the wiki bug tracker and a test case was introduced to the file of the specific C++ feature where the problem occurred.

3.3.2 Testing for Exceptions

The mechanism to test for exceptions is not quite obvious, so an example will be shown at this point.

The .rts test file may include the following syntax:

```
//@.config
fatalerror=true
```

Listing 3.9: Syntax to set variables inside a .rts file

The *fatalerror* variable may be retrieved using a member function of *RefactoringTest*:

```
@Override
protected void configureRefactoring(
    Properties refactoringProperties) {
```

```
  fatalError = Boolean.valueOf(
      refactoringProperties.getProperty(
      "fatalerror", "false")).booleanValue();
}
```

Listing 3.10: Accessing a property set in the .rts file

The *runTest* method may then assert that an error has occurred by using:

```
RefactoringStatus initialConditions =
    refactoring.checkInitialConditions(
    NULL_PROGRESS_MONITOR);
if (fatalError)
  assertConditionsFatalError(initialConditions);
```

Listing 3.11: Checking for errors inside the refactoring test class

All in all, the special refactoring test environment developed by [BG06] was a big help for relaxed refactoring.

3.3.3 Testing New File Creation

In case a member function is toggled from *in-header* to *in-implementation* and the implementation file does not exist, the user of the plug-in is asked through the `ToggleFileCreator` if he wants to create a new file and move the function there.

Long time it was not tested for this case and more than once this functionality was hurt and destroyed accidentally

In the .rts file, the `newfiles` variable has been introduced. This variable takes one or more file names separated by a comma.

```
//@.config
filename=A.h
newfiles=A.cpp, B.h, C.h
//@A.h
```

Listing 3.12: Syntax to set variables inside a .rts file

Further, there is no need to write the initial code state of the file, since it does not exist. However the final state must be written for comparison like in any other test.

```
//!FreefunctionFromHeaderToImpl
//#ch.hsr.ecl[...].ToggleRefactoringTest
//@.config
filename=A.h
newfiles=A.cpp
//@A.h
void /*$*/freefunction/*$$*/() {
    return;
}
//=
void freefunction();

//@A.cpp
//=


#include "A.h"

void freefunction()
{
    return;
}
```

Listing 3.13: Writing test for newfile creation

The files listed in the `newfiles` variable are deleted before the actual refactoring in the test is started. Then the new file gets created by the `ToggleFileCreator` (which functionality is specifically tested here) and gets compared with the expected source.

User Inputs

The easiest solution to test user inputs, being aware that it is not the nicest, is to mock the refactoring and to return an internal reference to the `ToggleRefactoringContext`. The context however has the ability to set predefined answer values to the question of the file creation, which is done in the test class.

3.3.4 Real World Test Environment

The toggle refactoring was tested with some open source projects found out in the wild.

COAST

The COAST [Hub10] source code was used as test environment for real-world tests as it uses a lot of C++ code features to test the toggling.

Toggling some functions in COAST, it was discovered that macros are not toggled correctly and are replaced by an `NullStatement` resulting in a function with a lot of semicolons. This was then fixed in later versions.

WebKit

Testing to toggle functions from Webkit [Web10] code showed us two problems.

First, Webkit uses a lot of namespaces. Until this point the Toggle plug-in did not work correctly with namespaces. Functions were moved completely out of the namespace and were referenced with the full qualified namespace as shown in listing 3.14. This is not very elegant.

```
#ifndef A_H_
#define A_H_

namespace N
class A {
  void function();
};
}

inline void N::A::function() {
  return;
}

#endif /* A_H_ */
```

Listing 3.14: `function()` with reference to namespace

The behavior was changed later to the following.

```cpp
#ifndef A_H_
#define A_H_

namespace N {
class A {
  void function();
};

inline void A::function() {
  return;
}

}

#endif /* A_H_ */
```

Listing 3.15: A.h, function definition in namespace

An other problem discovered with Webkit was that preprocessor statments were deleted. Have a look at section 4.2.5 for this problem.

3.3.5 Performance Tests

The simplest way to assess the speed of the refactoring is to look at the JUnit time measurements. The first test that is run takes more time and represents the time needed for first time toggling when the refactoring infrastructure has to be loaded.

All performance tests have been executed on the same developer machine, taking the average time of three consecutive runs of all tests. Five scenarios have been chosen to be able to observe the performance of the toggle refactoring:

1. First time toggling: Includes loading of the infrastructure and will take some more time.
2. Toggle from in class to header: Only one file is affected by this refactoring. This represents the least complex refactoring and should be the quickest one beside the reference test.

3. Toggle from implementation to header: Two files are affected here.
4. Emtpy reference test: A dummy refactoring that won't load and analyze any code. Shows what amount of time is consumed by the given refactoring infrastructure.

Another technique to measure time more accurately was checked out. For this, the *org.eclipse.test.performance* plug-in was used. This does not lead to satisfying results as stated in 4.5.2

4 Conclusions, Interpretation and Future Work

During the project, a lot of challenges have been discovered which were documented in the following sections along with a look back on the whole project and an outlook on what could be done in future theses.

4.1 Conclusions

4.1.1 Toggle Function Definition

The main goal of the project was to create a stable refactoring that would have a chance to be integrated into CDT. In the view of the authors, the developed plug-in became quite handy but should be tested by a larger community before it may be released to the public. One of the drawback is the issue with whitespaces which are not handled satisfactorily. See section 4.2.1 about newlines.

Anyhow, it should be taken into account that the C++ language specification (and its implementations by different compilers) may offer a lot more features than two developers could ever think of. It is not sure whether the covered special cases are enough general to cover all language constructs that may exist. Even programming against the C++ language specification is no guarantee that the refactoring will behave correctly out in the wild because compilers provide their own extensions and limitations.

All in all, the developer team is proud of the solution although aware of the fact that there may still be some improvements needed to satisfy a large audience.

4.1.2 Implement Function

Implement Function shares a lot of similarities and benefits of functionality developed for the *Toggle Function Definition*. This refactoring was developed in short time after the *Toggle Function Definition*.

4.1.3 Override Virtual Function

No deeper investigation on how this refactoring could benefit from the developed work has been done until now. This is still left to be implemented for another semester or bachelor thesis.

4.2 Known Issues

This section presents some unresolved problems or bugs which could not be fixed during the semester thesis.

4.2.1 Constructor / Destructor Bug

Problem: Let CDT create a new class with a constructor and a destructor. Then toggle the constructor out of the class definition. The Destructor will be overridden partially. This bug can also be triggered when a function above a constructor or a destructor is toggled. It seems that it is triggered with function names which do not have a type information and the `replace()` method of the ASTRewrite. There are some ways to prevent this bug, although it is not really a workaround. First the destructor can be made virtual. In the function above, arguments will also prevent this bug.

Cause: Unknown. It seems to be an offset bug and/or a rewriter bug.

Solution: Not yet solved.

4.2.2 Unneccessary Newlines

Problem: When toggling multiple times, a lot of newlines are generated by the rewriter.

Cause: Newlines are inserted by the rewriter before and after an new node but are not removed when removing the same node. To be able to judge how many newlines have to be inserted or removed, the whitespace situation around an affected node has to be analyzed thoroughly. Given figure 4.1 it could be tried to always remove one newline before and one newline after the removed function.

```
void before() {
}
// 1st newline added
void newFunction() {
} // 2nd newline added

void after() {
}
```

Listing 4.1: Whitespaces will not be removed blindly

Yet, it is not determined whether the programmer changed the code to look like in figure 4.2. There, it would be fatal to remove a character before and after the function because brackets would be removed instead.

```
void before(){}void newFunction(){}void after(){}
```

Listing 4.2: Code without the usual newlines

Workaround: First, the formatter could be used to remove multiple newlines. This breaks the programmers formatting which could be disruptive. Another solution is to manually change the generated text edits to avoid inserting or to delete more newlines. However, the changes are highly coupled to the different refactoring strategies. When this solution was tried to be implemented, it was a problem too that the generated text edits were changing their array positions, which made changes even more difficult. The resulting code was unstable and this solution is not recommended.

4.2.3 Comment Handling

Problem: A lot of freestanding comments are generated when toggling multiple times. These comments become leading if a function gets toggled below these freestanding comments.

Cause: When a node is removed by the rewriter, associated comments are not removed. This may be seen as a defensive strategy to avoid deleting comments accidentally. The ASTRewrite adopts all comments above a node as a leading comment not caring how many spaces or lines there are between node and comment.

Solution: No solution yet. This could be solved by a new rewriter.

4.2.4 Menu Integration (partially solved)

Problem: Adding a new menu item to the ”refactor“ menu is difficult when developing a separate plug-in.

Cause: Menu items are hardcoded inside *CRefactoringActionGroup*. No way was found to replace or change this class within a separate plug-in. In addition, the use of the *org.eclipse.ui.actionSets* extension point does not make inserting new items easier.

Workaround: The menu was added using *plugin.xml* and may be added by the user manually. See the manual in A.1.1 to solve this issue. Anyhow, the refactoring may always be invoked using the key binding of `Ctrl+Shift+V`.

4.2.5 Preprocessor Statements

Problem: If a preprocessor statement (e.g. *#ifdef*) is contained inside the parent of a rewritten, removed or inserted node, the preprocessor statement is deleted. Listing 4.3 shows an example where a class is rewritten and a contained preprocessor statement is removed as a side effect.

```
#ifdef EXAMPLE_H_                    // not affected
#define EXAMPLE_H_                   // not affected

class WillBeRewrittenImplicitly {
  #ifdef _X86__                      // will be removed
    void specificCode() {}           // will be removed
  #endif                             // will be removed

  void toBeManipulatedFunction(); // rewrite this
};

#endif                               // not affected
```

Listing 4.3: Jeopardized preprocessor statement inside a class

Cause: The rewriter does not support preprocessor statements.
Solution: None yet. This has to be solved by a fix for the rewriter which supports preprocessor statements.

There was a small workaround for this problem by warning about the presence of a preprocessor statement in the affected files. In the end this was dropped because this breaks the initial idea of a fast code flow.

4.2.6 Doxygen

Problem: Doxygen [Dox10] syntax `//!` may not be used in the test files since this syntax is used to specify the test name.
Cause: Refactoring tests need the `//!` syntax to specify the refactoring class which should be called for the selected code.
Solution: It was not looked for a solution to this problem. A solution could be to change the syntax for controlling the refactoring tests to something else.

4.2.7 Resource '/RegressionTestProject/A.h' does not Exist

Problem: When running refactoring test cases, a message randomly popped up:

```
!ENTRY org.eclipse.cdt.core 4 0 2010-12-13 ...
!MESSAGE Error: Resource '/RegressionTestProject/\
A.h' does not exist.
!STACK 1
org.eclipse.core.internal.resources.\
ResourceException: Resource '/RegressionTest\
Project/A.h' does not exist.
  at...ces.Resource.checkExists(Resource.java:326)
  [...]
```

Listing 4.4: Randomly appearing error message

Cause: Unknown (no deeper investigation)

Solution: Tests still pass without failure. It seemed this is no root of a problem. However it should be mentioned here.

4.3 Solved Issues

This section describes issues that have been resolved during this thesis.

4.3.1 Speed

Problem: Refactoring, especially the first run, was very slow in the beginning. Including a big header file slowed down the process even more.

Cause: The first thought was that header file indexing was the cause. However, the indexer option that skips already indexed headers is already enabled in `CRefactoring`. In the end, it was found out that most of the time was consumed by the `checkInitialConditions` method of `CRefactoring` that checked for problems inside the translation unit.

Solution: The super call to `checkInitialConditions` was omitted. This was possible since calling the super method is not necessary. For example, the translation unit provided by the `CRefactoring` is initialized here. But this is not used by the toggle plug-in because it uses smaller, per file translation units. Additionally the project files were indexed with options to prevent reindexing of already indexed files.

4.3.2 Accessing Standalone Header Files

Problem: Header files that are not included in any source file by default were not found by the indexer. Thus, it was not possible to analyze the source code of the affected header file.

Cause: By default, the indexer preference option `IndexerPreferences.KEY_INDEX_UNUSED_HEADERS_WITH_DEFAULT_LANG` is set to `false`. However, this option is needed for standalone header files to be indexed.

Solution: Set the described option in `IndexerPreferences` to `true`. This sets the indexer option per project since the index is retrieved per single project.

4.3.3 Indexing all Projects

Problem: Having multiple projects with the same file containing the same functions causes the plug-in to crash.

Cause: The index provided by the `CRefactoring` class returns an index containing all the indexes over all the open projects. This option only makes sense for refactorings like an "organize include statements"

Solution: Providing our own project local index and omitting functionality from CRefactoring class.

4.3.4 Selection

Problem: After toggling multiple times, the wrong functions were toggled or no selected function was found at all.

Cause: The region provided by `CRefactoring` pointed to a wrong code offset. This happens due to the fact that *IWorkbenchWindowActionDelegate*'s `selectionChanged` method is updated with outdated offset information.

Solution: The current selection is now based directly on the current active editor part's selection and fetched every time when toggling is started.

4.3.5 Fuzzy Whitespace Recognition

Problem: In past theses at HSR, the refactoring testing environment needed an exact definition of the generated code. This was annoying because same-looking code samples resulted in a red bar if white spaces were not the same. To make writing new tests easier, the comparison method was overridden to support fuzzy whitespace recognition.

Cause: The `TesterHelper` in the CDT test environment does compare the whole actual and expected code as `String` with [JUn10] `AssertEquals(String, String)`. This leads to a failing test as soon as a single whitespace is different between the expected and the actual code.

Solution: Leading whitespaces are recognized in both expected and actual source code. Then the smallest common divider is taken as as a tab length and replaced by a tabulator. Trailing superfluous newlines that are added by the ASTRewriter are ignored and also trailing whitespaces at the end of a line. After that the edited code is sent to the `assert()` for comparison.

4.3.6 Comments and Macros

Problem: Nodes inside a translation unit have to be copied to be changed since they are frozen. When nodes are copied, their surrounding comments get lost during rewrite[SZCF08]. This was annoying, since copying the function body provided a straightforward solution for replacing a declaration with a definition.

Another issue were macros. Macros are working perfectly when copied and rewritten inside the same translation unit. As soon as a macro is moved outside a translation unit, the macro will be expanded during rewrite or even deleted when no information about the macro is found.

Cause: The rewriter is using a method in `ASTCommenter` to get a `NodeCommentMap` of the rewritten translation unit. If a node is copied, it has another reference which will not be inside the comment map anymore. Thus, when the rewriter writes the new node, it will not notice that the node was replaced by another.

Solutions:

- Get the raw signature of the code parts that should be copied and insert them using an ASTLiteralNode.

  Pro: It works without changing the CDT core and macros are not expanded.

  Contra: Breaks indentation and inserts unneeded newlines. This solution was finally used. Afterwards, whitespace issues may be dealt with the formatter.

- Do as `ExtractFunction` does: rewrite each statement inside the function body separately.

  Pro: Automatic indentation.

  Contra: Touches the body although it does not need to be changed in any way.

- Change the CDT: Inside the `ChangeGenerator.generateChange`, the `NodeCommentMap` of the translation unit is fetched. By writing a patch, it was possible to insert new mappings into this map. This allowed to move comments of an old node to any newly created node.

Pro: Automatic indentation, developer may choose where to put the comments, every comment may be preserved.

Contra: Does not deal with macros, five classes need to be changed in CDT, comments need to be moved by hand. See the branch 'inject' inside the repository to study this solution. Due to intellectual property issues, this solution was not reviewed by the Institute For Software. To find out whether it is an acceptable solution the patch should be reviewed by the CDT community. Anyhow, a less disruptive solution was found for the problem.

- Find and insert comments by hand using an `IASTComment`.

  Pro: Lets the developer decide where to put the comment.

  Contra: Feature is commented-out in the 7.0.1 release of CDT, comments need to be moved by hand.

- Other solutions may be possible. An idea could be to register the comments whenever a node is being copied. Since `copy` is abstract in `IASTNode` and implemented separately inside every node, this would require a change inside every node class.

4.3.7 Toggling Function-Local Functions

Problem: When using function-local functions, the refactoring may produce code that won't compile.

Cause: Despite in the C++ standard[30910] function-local functions are not allowed, the GNU C compiler allows to define such nested functions[Fre10]. In this case the selection detection finds the nested function if selected and it is tried to toggle it. However, it is not guaranteed that valid code will be generated.

Solution: Toggling is disabled for such nested function definitions.

4.4 Future Work

The toggle refactoring was developed as a separate plug-in so integration into the CDT project should be possible if desired.

It should be a small task to provide a solution for multi-toggling. If the user selects more than one definition, all of them could be toggled. An example workflow could be "Create new class (inherit from an abstract class)", "add unimplemented methods", "toggle all methods to an implementation file".

A big problem in this refactoring was the rewriter. Since it is limited in its functionality by limited support for comments, no preprocessor statement support, inserting newlines over and over, and even producing wrong code with the constructor bug (see section 4.2.1), it is extremely time consuming to find workarounds for such problems. Therefore before any new refactoring is developed, the rewriter should be fixed or rewritten. As in the last meeting of this semester thesis with the supervisor, it was ensured such a work will be done by the Institut fuer Software.

A mechanism could be implemented that fixes indentation after refactoring. Better user feedback in case of errors could be provided.

At the end there was no time left to do the *Override Virtual Member Function* refactoring. This is still left as a semester thesis.

4.5 Interpretation

After implementation, a personal look backwards was made on what the resulting refactoring is capable of and what may still need some improvement.

4.5.1 Features and Limitations

Toggling functions is available inside any class or namespace hierarchy and may be invoked when the selection is anywhere inside a function declaration or body. Basic templated functions are supported as well. However, there may be template special cases that we have not thought of.

On the other side, the code generator removes preprocessor statements that stay inside a node that has to be rewritten. Removing comments of removed nodes was not achieved without changing the rewriter since this is a bug in the rewriter.

4.5.2 Performance Results

It is difficult to compare the speed with other refactorings of CDT since wizards are used for the other known refactorings. However, the goal was reached that the refactoring is executing almost instantly.

It was planned to measure the speed of the JUnit tests as explained in section 3.3.5. However, the displayed time does not represent the actual speed of the refactoring. This may be due to the fact that tests are not being invoked exactly the same way as the actual refactoring. The manually invoked refactoring is run with the help of `ToggleRefactoringRunner`.

The results from the *org.eclipse.test.performance* speed tests were not used either. Since in reality the refactorings are much slower than the (repeated) measurements, resulting values may only be compared relative to each other.

In the end, the only way to judge whether the refactorings became quicker is to check out an older version and to try it out manually. Included libraries like iostream slowed down the refactoring noticeably before speed was improved.

4.5.3 Personal Review

Some words from the authors about the developed plug-in, project management, of what was fun and what not.

Martin Schwab

What I like about the developed refactoring is that it was possible to implement it without a wizard. However, the user needs full trust in the code generation that it will not break code and this is currently not given when preprocessor statements are used. Nevertheless I am glad the developed plug-in is able to write complex template definitions that I could not write myself without the C++ specification by my side. This could save a lot of time and hassle for programmers.

What next? If the plug-in is integrated into CDT, I would be interested in the Eclipse Usage Data Collector [usa10] to check whether users find out when and how they can profit from the refactoring and whether they use it repeatedly. If the refactoring is considered helpful by users and applied as it is designed to use, this should be reflected in a high execution count compared to the user count.

Thomas Kallenberg

Personally I like the C++ programming language. Despite the complexity it can be an alternative to Java or other OOP languages. Specially if there is a focus on performance or other subjects where templates do fit nicely in the concept of the project. The one touch toggle refactoring is another step towards a better knowledge of C++.

What I liked about the project was the clear and productive communication with the supervisor. The environment to develop a CDT refactoring provided enough functionality to develop a good refactoring in the given time scope but let us enough freedom to realize innovative ideas.

The goal was to develop a plug-in that is used by the world and not thrown away. Even if we could not solve all issues, specially with comments and macros, I think we achieved the main task and realized our supervisors and customers idea of a toggle definition.

What we Would do the Same Way

Using Git[GIT10] for version control was very useful. The provided development server was occasionally down during the first weeks and it was possible to continue work locally with version control. Being able to work locally was also helpful to work on the train.

Working next to each other in the same study room was helpful to get quick answers for questions, reduce slow written communication and playing a round of table foot when concentration was used up.

For each meeting, the planned tasks were collected inside the agenda, then rubber-stamped by the supervisor and transformed into issues for the following week. This way, a minimal administrative overhead was produced.

What we Would not do the Same Way

Time Management

During most of the project, time was tracked for every issue. However, the collected data was not actively used to measure team velocity and estimate further issues. This valuable data could have been used for better scope prediction.

Commits not Connected with Issues

During the project we sometimes “forgot” to track our time. Sometimes it was forgotten and sometimes it was delayed (and then forgotten) to speed up implementation of the functionality. This is clearly not a good behavior since the danger is high to forget about the tracked time as it happened to us many times.

As an improvements in the bachelor thesis we thought about connecting every commit to the version control with an issue. It could even be checked with a server side script during committing. If the commit message does not contain a valid issue number the commit is rejected. This forces the developer to connect his commit to an issue.

Wiki and Documentation Differences

The special cases that were listed on the project wiki were useful to communicate but it may have saved time if they were directly integrated into the documentation.

Writing Documentation at the End

We knew about the fact that writing documentation at the end is a hard thing due to time shortage. We created the documentation structure early and started to write some text to it. But somewhere in the middle of the project we did not kept in mind to continue this. This and changes to the code until the end led to a lot of stress in the last two weeks since the documentation was not written as far as expected.

Chapters like specification should be written early. The wiki is not read, but the documentation is read and if written early mistakes do not live long, as long as the documentation is reviewed continuously.

Glossary should be written continuously and even some parts of the implementation needs to be documented early.

Redmine Fine Tuning

The default Redmine *trackers* and *categories* that were used were not sufficient to track time in a way that shiny charts could be produced for categories like "implementation", "documentation" and "administration. For future projects we need more fine grained categories and trackers.

Multiple Git Branches

While using one master git branch for every developer, a second git branch called development was introduced. During half a week new work was committed to the development branch and then both trees were merged. The idea behind this was to have always a stable master branch. This allowed to carelessly mess around inside the development branch which resulted in nobody daring to pull from the others branch. Inside the master branch everything was already merged and the development branch was in an "always unstable" condition. If something unstable is introduced to the repository one should explicitly use a separate branch and include this in the master as soon as possible.

A User Manual

To quickly have a look at what was realized during the semester thesis, just install the *Toggle Function Definition* code automation according to this manual.

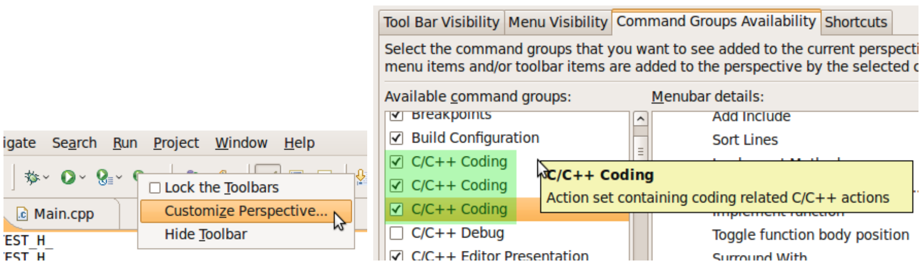
A.1 Installation of the Plug-in

To try out the plug-in, you will need a running Eclipse Helios (3.6) installation with CDT 7.0.1 already installed. Choose ”Help”, ”Install New Software...” and type in the following url into the address bar:

`http://sinv-56042.edu.hsr.ch/updatesite`

The plug-in may now be installed using the install wizard. Be aware that the update site is hosted on a virtual server that will be removed at the end of summer 2011.

A.1.1 Activate the Menu Item


Select the command groups that you want to see added to the current perspecti
menu items and/or toolbar items are added to the perspective by the selected c

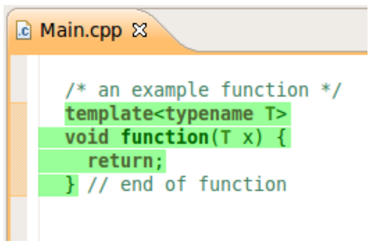
C/C++ Coding
Action set containing coding related C/C++ actions

Figure A.1: To use the refactoring from the menu, it needs to be enabled manually after installation.

To run the refactoring using the menu, some changes have to be applied to the current perspective. Right-click on the toolbar and choose ”Customize perspective...”. Then go to the ”Groups and commands visibility" tab and check **all** "C++ coding" boxes. The menu "Toggle Function Definition" should now be visible inside the source menu.

A.2 Using the Refactoring

Toggling is available whenever the cursor is inside a function declaration or definition. Any selection between the first and the last character of the function definition (without comments) is considered valid for toggling. Figure A.2 depicts all valid selection ranges in an example code. As soon as the cursor is inside the valid range,

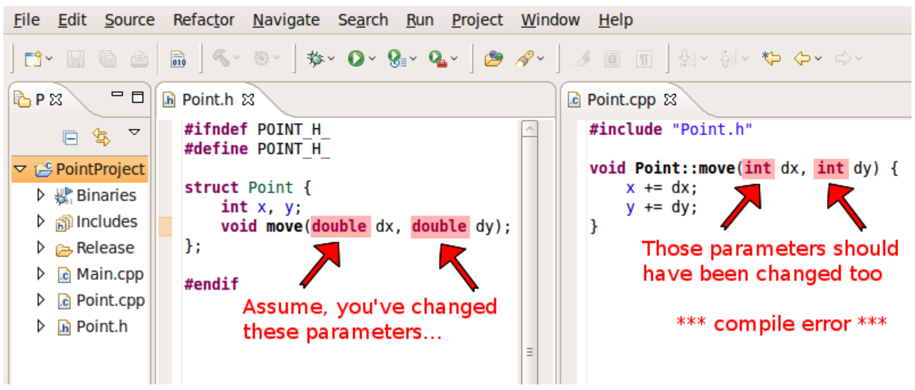


```
/* an example function */
template<typename T>
void function(T x) {
    return;
} // end of function
```

Figure A.2: Region for valid selection positions

toggling may be invoked by pressing `Ctrl+Shift+v`.

A.3 Example



```
#ifndef POINT_H_
#define POINT_H_

struct Point {
    int x, y;
    void move(double dx, double dy);
};

#endif
```

```
#include "Point.h"

void Point::move(int dx, int dy) {
    x += dx;
    y += dy;
}
```

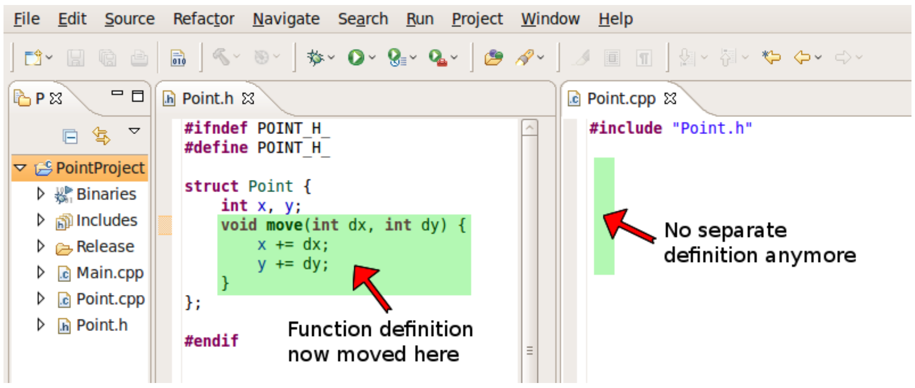
Figure A.3: Redundant declarations that were not both updated

Imagine the situation in figure A.3 where a programmer forgot to update a function signature of a function that was defined inside another file. After finding and correcting the signatures, the programmer would like to define the function directly inside the header file, removing the redundancy of a separate definition. What would he do?

Perhaps he would jump to the definition, copy its body, remove the definition, jump back to the declaration, remove its semicolon, add curly brackets and paste the copied function body. All in all, seven actions.

A.3.1 The Solution

This is the point where *Toggle Function Body* makes life easier. Imagine the same situation (with correct signatures though) and the user presses the key combination `Ctrl+Shift+V`. Figure A.4 shows what happens to the code. The declaration got automatically replaced by the definition.


File Edit Source Refactor Navigate Search Run Project Window Help
P
PointProject
Binaries
Includes
Release
Main.cpp
Point.cpp
Point.h
Point.h
Point.cpp
Function definition now moved here
No separate definition anymore

```
#ifndef POINT_H_
#define POINT_H_

struct Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
};

#endif
```

```
#include "Point.h"
```

Figure A.4: By the touch of a key combination, the function definition is moved

The function is now defined in one place and its signature may be changed without updating the declaration separately. Seven actions have been replaced by just one keystroke. Convinced?

B Project Setup

Project setup needed two weeks to become stable. In this chapter, solutions are described that may help future projects to build up their project environment more efficiently.

B.1 Configuration Management

This section describes what techniques were used throughout this project and what product versions were needed to realize an agile production environment for Eclipse plug-in development.

B.1.1 Project Server

As a project server Ubuntu Linux 10.04.1 LTS (Lucid Lynx) on a VmWare Cluster was used.

B.1.2 Git

As a version control system, Git[GIT10] was used. This time it was not used like an SVN[SVN10] replacement but instead to get some redundancy by storing the source code on multiple servers. Both project developers had their "own" git server on which the developer committed. It was merged between these servers and then pushed to the main repository for automatic testing.

Later in the project, a master and a development branch have been introduced to improve trust for the master branch.

B.1.3 Maven and Tycho

Tycho[tyc10] is a plug-in for Maven[mav10] v3.0 to build an Eclipse plug-in and to execute its tests. Maven3 is required for this to work. Although Maven3 is beta, it was proven stable during the project.

B.1.4 Continuous Integration

Hudson is a build server for executing repeated build jobs. It is specialized for executing Java projects. To get the tests executed on the fake X sever, the `DISPLAY` environment variable must be set. If not, tests will fail with a cryptic SWTError.

There is a plug-in for Hudson[hud10] to set the environment variable to the right value. In most cases this is **:0**.

B.2 Project Management

Redmine[red10] was used to track issues, milestones and agendas and as an information radar for the supervisor.

The Redmine version used crashed every now and then. This issue went away after some memory upgrade. Redmine depends how it is configured. Using the passenger ruby module made it quite stable.

B.3 Test Environment

To execute the tests on a headless server with Hudson build server first a fake X Server was needed. Xvfb[xvf10] was used for the job. Maven has to be explicitly told which test class to execute and in which folder the test class is located. If this is not done properly the tests will fail.

B.3.1 Test Coverage

Creating new files was not tested in the beginning due to the dialog box popping up during execution. However, with the help of the same technique that was used to test for errors, the problem could be solved. Before a dialog box is popped up asking whether to create a new file, it is checked whether the test framework has set a flag to skip that question.

The code coverage tool EclEmma[ecl10] was a good help to find dead code and to think about justification of code blocks.

B.3.2 Documentation

The documentation was originally based on [AV08] and adapted to meet the requirements of the project.

C Time Management

This chapter should help other students guessing the amount of work that is involved in writing a refactoring semester thesis.

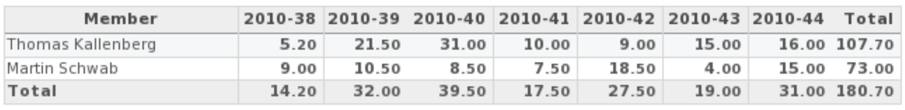
C.1 Time Budget

The extent of work expected for a semester thesis at HSR is 8 ECTS which corresponds to an effort of 240 hours per person. Issues were created after the meetings on Thursday and an amount of time was guessed for them.

As mentioned in the section 4.5.3 we sometimes forgot to log the time right away. Later we guessed the time invested in the issues we worked on.

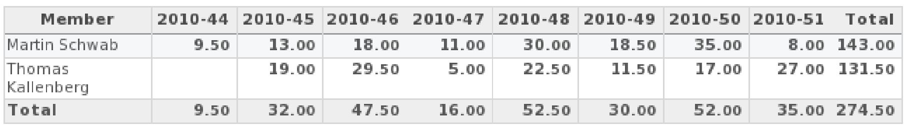
C.2 What the Charts Tell us

This tables illustrate the workload by week and team member.



Member	2010-38	2010-39	2010-40	2010-41	2010-42	2010-43	2010-44	Total
Thomas Kallenberg	5.20	21.50	31.00	10.00	9.00	15.00	16.00	107.70
Martin Schwab	9.00	10.50	8.50	7.50	18.50	4.00	15.00	73.00
Total	14.20	32.00	39.50	17.50	27.50	19.00	31.00	180.70

Figure C.1: Working hours of the team members by weeks from beginning of the project to 11th of November



Member	2010-44	2010-45	2010-46	2010-47	2010-48	2010-49	2010-50	2010-51	Total
Martin Schwab	9.50	13.00	18.00	11.00	30.00	18.50	35.00	8.00	143.00
Thomas Kallenberg		19.00	29.50	5.00	22.50	11.50	17.00	27.00	131.50
Total	9.50	32.00	47.50	16.00	52.50	30.00	52.00	35.00	274.50

Figure C.2: Working hours of the team members by weeks from 11th November to 23rd of December

Summing up all, results in about 450 hours worked for the project.

C.3 What the Team is Saying

Charts are generated out of numbers which have been typed in by humans. There has been a time between week eight and ten where time tracking was abandoned and added three weeks later. (Also see section 4.5.3). Also, there were a lot of discussions that could not have been assigned to a specific issue. Not every minute was tracked, so be aware that the actual spent time was about ten percent above the reported time.

D Glossary

- **AST**: Abstract Syntax Tree
- **CDT**: C/C++ Development Tooling for Eclipse
- **Content Assist**: Content assist allows you to provide context sensitive content completion upon user request. Popup windows are used to propose possible text choices to complete a phrase. The user can select these choices for insertion in the text. [ass10]
- **Declaration**: "A declaration introduces names into a translation unit or redeclares names introduced by previous declarations. A declaration specifies the interpretation and attributes of these names."[30910]
- **Definition**: "A declaration is a definition unless it declares a function without specifying the function's body, it contains the extern specifier or a linkage-specification and neither an initializer nor a function-body, it declares a static data member in a class definition, it is a class name declaration, it is an opaque-enum-declaration, or it is a typedef declaration, a using-declaration, or a using-directive."[30910]
- **Doxygen**: A documentation system that supports multiple programming languages.
- **Free function**: Whenever free functions are mentioned throughout this document, non-member functions are meant in the C++ terminology. These are functions which are not part of a structure or a class.
- **Header file**: A file with the file extensions .h, .hpp or .hxx
- **Non-member function**: A function that does not belong to any type.

- **Problem node**: If a source code range has syntactical errors, the CDT parser wraps it into an `IASTProblem` which can be handled as if it were a normal `IASTNode`.
- **.rts file**: Before/after tests for refactorings may be written inside a file with the extension ".rts" using a special syntax. CDT offers a mechanism to automatically read those files, run the refactoring and compare the sources.
- **Source file**: A file with the file extensions .c, .cpp or .cxx
- **Toggle refactoring**: The developed *Toggle Function Definition* code automation was referred to by this name because it is shorter.
- **Toggling**: The act of invoking the plug-in developed during this project to move a function definition to another place.
- **Translation unit**: "The text of the program is kept in units called *source files* [...]. A source file together with all the headers and source files included via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a *translation unit.*"[30910]

Bibliography

[30910] ISO/IEC JTC1 SC22 WG21 N 3092. *IS 14882: Programming Languages – C++*. International Organization for Standardization, Geneva, Switzerland, March 2010.

[ass10] Content Assist. `http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm`, 2010. [Online; accessed 20-December-2010].

[AV08] A. Simeon A. Verhein. Werkzeugkasten Technische Berichte 1, 2008.

[BG06] Leo Büttiker and Emanuel Graf. C++ refactoring support für eclipse-cdt. `http://ifsoftware.ch/uploads/tx_icscrm/i_da_2006_refactoring_support_fuer_eclipse_cdt_leo_buettiker_emanuel_graf.pdf`, 2006. [Online; accessed 13-December-2010].

[Dox10] Doxygen – Generate documentation from source code. `http://www.doxygen.org/`, 2010. [Online; accessed 21-December-2010].

[ecl10] EclEmma – Java Code Coverage for Eclipse (Version 1.5.1). `http://www.eclemma.org`, 2010. [Online; accessed 17-December-2010].

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Object Technology Series. Addison Wesley, October 1999.

[Fre06] Frenzel, Leif. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. *Eclipse Magazin*, 5, January 2006. see `http://www.eclipse.org/articles/Article-LTK/ltk.html`.

[Fre10] Free Software Foundation. *Nested Functions - Using the GNU Compiler Collection (GCC)*, 2010. [Online; accessed 24-December-2010].

[GIT10] Download page for Git version control system (Version 1.7.0). `http://www.kernel.org/pub/software/scm/git/`, 2010. [Online; accessed 16-December-2010].

[Hub10] Marcel Huber. Coast – C++ Open Application Server Toolkit. personal communication, 2010. also known as WebDisplay2 `http://wiki.hsr.ch/APF/files/WebDisplay2_Architecture.pdf`.

[hud10] Hudson – Extensible continuous integration server (Version 1.352). `http://hudson-ci.org/`, 2010. [Online; accessed 17-December-2010].

[JUn10] JUnit.org – Resources for Test Driven Development. `http://www.junit.org/`, 2010. [Online; accessed 21-December-2010].

[mav10] Apache Maven (Version 3.0-beta3). `http://maven.apache.org/`, 2010. [Online; accessed 16-December-2010].

[red10] Redmine – Project Management Web Application (Version 0.9.3-1). `http://www.redmine.org/`, 2010. [Online; accessed 22-December-2010].

[SP03] Herb Sutter and Tom Plum. Why We Can't Afford Export. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf`, March 2003.

[SVN10] Apache Subversion. `http://subversion.apache.org/`, 2010. [Online; accessed 22-December-2010].

[SZCF08] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 653–662, New York, NY, USA, 2008. ACM.

[tyc10] Tycho – Sonatype (Version part of Maven 3). http://tycho.sonatype.org/, 2010. [Online; accessed 17-December-2010].

[usa10] Usage Data Collector Results. http://www.eclipse.org/org/usagedata/results.php, 2010. [Online; accessed 16-December-2010].

[Web10] The WebKit Open Source Project (Version SVN head: r71799). http://webkit.org/, 2010. [Online; accessed 22-December-2010].

[xvf10] Xvfb – virtual framebuffer X server for X Version 11. http://www.x.org/archive/X11R6.8.1/doc/Xvfb.1.html, 2010. [Online; accessed 22-December-2010].