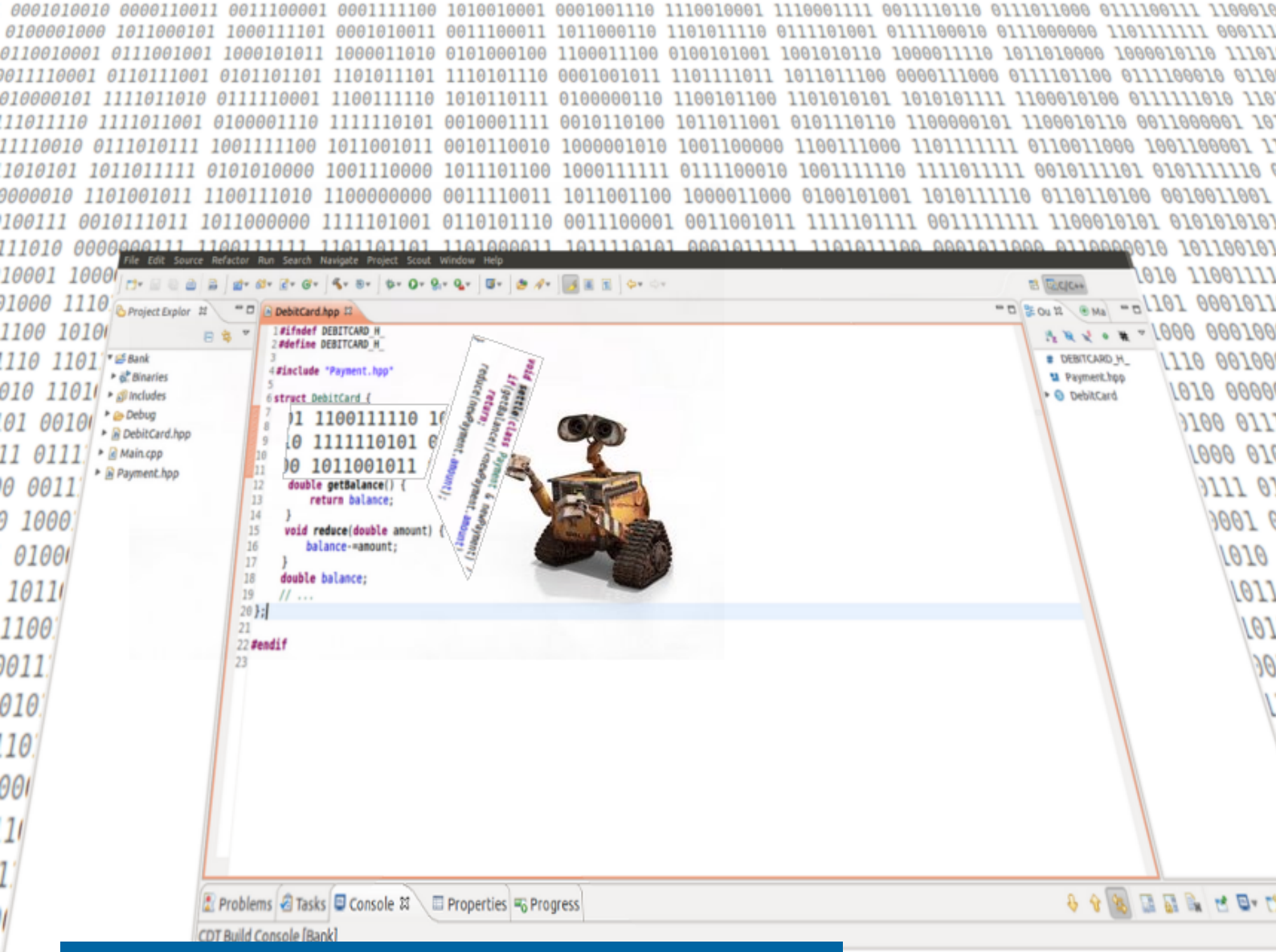
Master-Thesis, Fall Semester 2011

TurboMove

Move Refactorings for Eclipse CDT



IFS INSTITUTE FOR SOFTWARE

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

FHO Fachhochschule Ostschweiz

Author: Yves Thrier

Advisor: Peter Sommerlad

Abstract

While writing code for a project, a programmer has to think about where to put that code. The initial decision of placing it to the chosen location is not always correct or the appropriate position changes as the code-base evolves. A class may obtain too much behaviour over time which could be isolated. Or a member-function uses more functionality from its parameter than the instance itself. Therefore, moving the code related to such problems to a more suitable position helps to lower coupling and increases the quality, understandability and maintainability of source-code. In C++, determining the correct position for functionality is harder compared to other programming languages, due to the separation into declarations and definitions and by allowing free functions. This also makes it hard to create refactorings offering code moving in an automated fashion. This master-thesis overcomes these problems by introducing a separation between a logical and a physical move. In addition, it tackles different types of code relocation in a C++ project. There are transformations allowing to move member functions among classes, into another file or to convert them into free functions. In addition, it is possible to change the namespace membership of types and free functions by altering them to become members of the parent namespace. While all these move refactorings retain compilability, they also provide different configuration options to customise the style of adjusting visibilities and call-sites. The transformations are available as a plug-in for Eclipse CDT.

Management Summary

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

Motivation

Move refactorings are an important group of code transformations. A software engineer writing code must always think about where to put that code. However, the decision to place the functionality to the chosen location is not always correct or changes as the code-base evolves. For example, a class may have obtained too much behaviour over time which could be isolated, or a member function is better suited to be a free function. In C++, it can be very hard to decide on the appropriate location for adding functionality due to the separation between declarations and definitions and other options such as implementing a free function instead of a member function. Thus, creating refactorings performing such transformations in an automated fashion is difficult. However, doing such moves manually is tedious and error-prone, therefore, often avoided resulting in less than optimal structured code. Thus, having refactorings for move transformations is desirable and helps to make source-code easier to understand and maintain.

Goals

In this master-thesis, we tried to tackle these issues. The goal of the project was to evaluate possible move refactorings for C++ and to create an approach to apply these transformations in an automated fashion. A set of the most valuable moves had to be implemented including resolving potential conflicts and dependencies such as call-sites and visibilities. The compilability of the code must be retained as well. In addition, a facility to recognise code-locations with move potential and give a move proposal for them should have been developed. The transformations must be implemented as an Eclipse *CDT* plug-in.

Results

TurboMove is an Eclipse *CDT* plug-in capable of performing four different move refactorings. The transformations are applying the concept of separating the moves into a logical and a physical relocation. In addition, the applied transformations try to reduce the amount of changes to the input to a minimum. This reduces the complexity of the refactorings and make them easier to use.

The first refactoring allows a user to move a member-function to another class. The declaration of the member-function is relocated to belong to the new type, thus, beeing a logical membership change. The definition retains the physical position, but is changed to belong to the new type. All call-sites of the member-function are changed to be invoked on the new target type. In addition, dependencies to the originating class are satisfied by adding a new parameter of the originating type. A user can select to keep the original function as a delegate

to avoid call-site changes and to adjust visibilities of the originating type using labels or by adding a function friend declaration.

```
/**================================================
Example of moving a member-function to another class
================================================**/
// Before Transformation
struct Origin {
  void foo() {
    bar();
  }
  void bar() {
    // ...
  }
};

struct Destination {
  // ...
};

// After Transformation
struct Origin {
  void bar() {
    // ...
  }
};

struct Destination {
  void foo(Origin & newOrigin) {
    newOrigin.bar();
  }
  // ...
};
```

Listing (1) *Moving a Member-Function to Another Class*

As a complement to moving a member-function to another class, a transformation to relocate a member-function definition into a new or existing file was created. By moving copies of the include statements present in the originating file to the destination file, it is ensured that the required elements for the member-function are visible at the destination

The third implemented refactoring converts a member-function into a free-function. An additional parameter is added to solve dependencies to the originating type. The call-sites are changed to use the free-function instead of the previous member-function as well. In addition, a user can chose to add either a function friend declaration or visibility labels to adjust visibilites for required elements in the originating type.

The final refactoring created changes the namespace membership of a free-function or a type

to the parent namespace in the current hierarchy. The call-sites are changed if *Argument Dependent Lookup* (*ADL*) is not or no longer sufficient. Optionally, a user can chose to add a using declaration for the moved type or free-function in the originating namespace to avoid call-site adaption.

```
1 /**================================================
2 Example of moving a free-function to the parent
3 namespace
4 ================================================**/
5 // Before Transformation
6 namespace A {
7   void before();
8   void foobar();
9   void after();
10 }
11
12 // After Transformation
13 namespace A {
14   void before();
15 }
16 void foobar();
17 namespace A {
18   void after();
19 }
```

Listing (2) *Moving a Free-Function to the Parent Namespace*

All refactorings provide a user interface to Eclipse from which a user can control the refactoring to perform and the associated options.

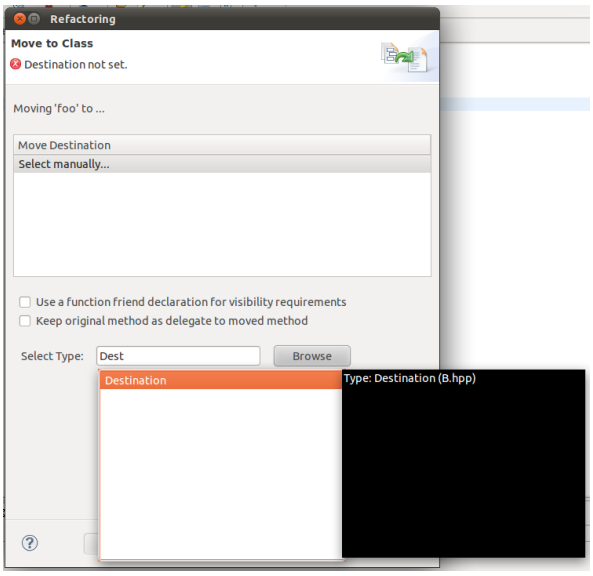


Figure (1) *TurboMove User Interface Example*

Future Work

The TurboMove plug-in covers only a small part of all evaluated move refactorings. Based on the analysis made in this thesis, additional move refactorings such as the complement for *move type or free-function up to parent namespace* could be implemented. Also, none of the variable based moves could be implemented in this project due to time issues and focusing on other move refactorings, hence, they may serve as a good extension for this plug-in.

Since it was not possible to add a facility to recognise candidates for a move, future work could include to add Codan [Ecl12c] support to the TurboMove plug-in. By giving immediate feedback to a user with markers in Eclipse, the generated move proposals could be used to perform the desired transformation without the need for a user interface.

Contents

1 Introduction

This introduction Chapter provides an overview of why move refactorings are important for writing excellent, understandable and maintainable code. Next to this, we give an overview of the current situation regarding move refactorings in Eclipse *CDT*. We conclude with the project goals for this master-thesis in the final Section.

1.1 Motivation

Move refactorings tackle the fundamental problem in software engineering of assigning responsibilities. In object-oriented programming languages, these responsibilities are mostly related to the question of where to put a method or function, or in the design of class hierarchies. Assigning these responsibilities is also rather an iterativ process and is not done once for each piece of code. When we assign responsibilities we have to deal with some challenges:

Assigning Responsibilities is Hard

The first time you write a piece of code, it seems obvious to you to put that code to class A. But later you recognise, that it suites better to class B. Do you change the code manually, or do you proceed your work ignoring this potential design flaw?

Responsibilities can change

Changing requirements can induce changes to the source-code. This can also rise the need to rethink the assigned responsibilities. Applying both parts of these changes is fundamental to sustain software quality and accomplish the requirement goals.

Emerging Design

Software is written step-by-step, incrementally introducing new functionality and features. It is possible that due to this responsibilities change or should be moved to another level of abstraction. Thus, updating responsibilities is important in day-to-day software development.

This list of challenges is not complete, but the importance of move refactorings is clearly evident. If we think of reassigning or changing responsibilities in our source-code, we quickly see that doing this manually is difficult. There is more work included than simply cut&paste the code to the new location. Dependent code have to be maintained making sure the changes do not break compilability. For example, moving a member-function from class A to class B requires changes at the call-sites of this member-function and we may even need an instance of the class B, or access to such an instance. Doing all the necessary work by hand is slow, awkward and error-prone. An automated way to apply such refactorings cannot only save time, it can improve the quality of your code and ensure a reasonable output state as well.

1.2 Actual Situation

The Eclipse *CDT IDE* does not have any support for move refactorings up-to-date. As we have seen in our motivation, move refactorings are clearly an important group of refactorings and this lack of functionality in the Eclipse *CDT* is a serious drawback for the acceptance of

Eclipse to be a suitable alternative to other *IDE*s for C++ development. In addition, move refactorings are likely to be harder to apply in C++ compared to for example Java. The complex nature of C++ introduces more challenges, but also more opportunities for viable moves as well. With this master-thesis, we tackle the absence of move refactorings in Eclipse *CDT*.

1.3 Project Goals

The goal of this master-thesis is to analyse possible move refactorings in the C++ programming language along with the challenges and related problems these transformations introduce. While existing work such as [Opd92] handles move refactorings, it does on a different level of abstraction. The evaluated moves should be implemented as an Eclipse *CDT* refactoring plug-in, capable of applying the move mechanics and solving the related dependencies and conflicts. Due to the complex nature of C++ it is likely to find more move refactorings in the analysis than can be actually implemented. The focus is on the realisation of the most valuable transformations found. The implemented refactorings must preserve the original semantics and retain compilability. In addition, a facility to automatically recognise potential moveable code along with proposing a desirable move for this code should be developed.

2 Analysis

In this section, we provide an analysis of different types of move refactorings. First, an overview of the move refactorings separated in categories based on the code to move is given. Afterwards the set of moves to implement in this master-thesis is described with a in-depth analysis of the selected moves.

2.1 Types of Move Refactorings

Move refactorings come in different flavours and difficulties. An excellent starting point for move refactorings can be found in the refactoring book of Martin Folwer [Fow04]. He described eight types of move refactorings, such as *move method*, *extract class*, *inline class*, etc. These list of refactorings allows us to identify a key point of move refactorings: Most move refactorings or rather most refactorings have a counterpart. If we apply move method to move a method from A to B, we can apply move method again to move the same method back from B to A. In this case, the refactoring itself is it's counterpart. In contrast, the counterpart for extract class is inline class, and vice versa.
As a secondary source of information for move refactorings, Joshua Kerievsky's Book *Refactoring to Patterns* [Ker04] is very helpful. But apparently, they are harder to introduce in an automated fashion compared to the "low-level" move refactorings described by Martin Fowler. Refactoring to a pattern requires more work to do and a pattern is not always applied the same way. Thus a software to do this automatically would have to either "guess" what to do, do it only in one way or likely require an incredible amount of user input to perform correctly. It is not impossible to do, however, it seems not reasonable to do in the current situation of the Eclipse *CDT IDE*, where no move refactoring support exists at all. Hence this project will focus on the more valuable "low-level" move refactorings of Martin Fowler applied to C++.

The foundation of a move refactoring contains of three key factors. There is a *move source*. This is the *"What to move?"* of the refactoring. The second factor is the *move destination* asking *"Where to move?"*. Number three tackles the "What is missing?" meaning the dependencies that must be solved. These additional challenges can be classified into different categories:

Outside Dependencies

If a move refactoring takes place in an automated fashion, the usage-sites related to the moved element must be adjusted accordingly. For example, a member-function moved from class A to class B must be invoked on B after the move and can no longer be used from instances of A. This type of dependency also relates to required definitions for using the element, such as arguments of a function or the target type definition. We name this type of dependencies *outside dependencies*.

```
/**================================================
Example of changing the call-site of a member-
function in a move to another type refactoring
================================================**/
// Before Transformation
int main() {
  // ...
  Origin a;
  Destination b;
  // Invoking foo on 'a' with argument 'b'
  a.foo(b);
}

// After Transformation
int main() {
  // ...
  Origin a;
  Destination b;
  // Change invocation owner to 'b' with argument 'a'
  b.foo(a);
}
```

Listing (3) *Example of an Outside-Dependency*

Inside Dependencies

In contrast to the outside dependencies, it is possible that other dependencies exist. For example, the moved member-function of class A to class B used a member-variable of class A. These *inside dependencies* have to be resolved as well. An inside dependencies may also include required definitions that must be available at the new location of the moved element, ensuring compilability of the source-code.

Conflict Resolving

A move can introduce a conflict making a move impossible, or only allowed under certain situations or configurations. For example, moving a member-function M from class A to class B with an existing member-function M can yield a conflict in case the overload resolution yields both functions if arguments are applied.

```
/**================================================
Example of conflicting function overloads if a
move is applied
================================================**/
// Conflicting Functions
struct Origin {
  void foo(int i, std::vector<int> v) {
    // ...
  }
};

struct Destination {
  // Function with same name and overload exists
  void foo(int i, std::vector<int> v) {
    // ...
  }
};

// No Conflicting Functions
struct Origin {
  void foo(int i, std::vector<double> v) {
    // ...
  }
};

struct Destination {
  void foo(int i, std::vector<int> v) {
    // ...
  }
};
```

Listing (4) *Example of a Move Conflict*

C++ Language Rules

Moving source-code can introduce violations of the C++ language rules. For example, moving an operator implemented as a class member-function to a free-function is not allowed for specific operators (e.g., the assignment operator must be a nonstatic member-function).

It is noteworthy that not every category may yield a problem to solve for a specific move refactoring. In addition an instance of a particular move refactoring may introduce additional problems not related to these categories at all. However, they establish a vocabulary to think about the problems for potential move refactorings. With these informations, we are now prepared to define different types of move refactorings.

2.1.1 Function Based Moves

These types of move refactorings tackle the problem of moving macros, operators, member- and free-functions. Please note that we will not make a separation between the move of a declaration or a definition for this list of move refactorings at this time (this problem will be investigated as we analyse the selected move refactorings in depth), with one exception, namely a separate or merge refactoring for declaration and definition. Following a list of possible move refactorings related to functions.

Refactoring	**Reverse Operation**
(1) Member-Function to Free-Function	(2) Free-Function to Member-Function
(2) Free-Function to Member-Function	(1) Member-Function to Free-Function
(3) Pull-Up Member-Function to Parent Class	(4) Push-Down Member-Function to Child Class
(4) Push-Down Member-Function to Child Class	(3) Pull-Up Member-Function to Parent Class
(5) Move Member-Function to another Class	(5) Move Member-Function to another Class
(6) Move Free-Function to another File	(6) Move Free-Function to another File
(7) Merge Member-/Free-Function Definition and Declaration	(8) Separate Member-/Free-Function Definition and Declaration
(8) Separate Member-/Free-Function Definition and Declaration	(7) Merge Member-/Free-Function Definition and Declaration
(9) Move Free-Function to a New/Existing Namespace	(9) Move Free-Function to a New/Existing Namespace
(10) Move Macro Definition to New/Existing File	(10) Move Macro Definition to New/Existing File

It is important to state that *(3) Pull-Up Member-Function to Parent Class* and *(4) Push-Down Member-Function to Child Class* are not that important in C++ compared to, for example, Java, due to the fact that C++ offers convenient ways to avoid inheritance while keeping flexibility with templates. Converting existing types and functions into a template was also already developed in [Thr10]. The merge and separate definition and declaration move refactoring is already implemented and integrated in the Eclipse *CDT* Indigo release [Sch10] as ”toggle” refactoring, however, they are stated here for completeness.

Related Problems

Outside-Dependencies

- Missing class-instances or instance-access is insufficient for function call-site adjustment
- Visibility of classes not sufficient for function call-site adjustment
- Parameter, return type and/or types used in function body not known at the new position

Some of the stated outside-dependencies are not obvious, therefore, we will try to support them with further explanations. The problem of missing class-instances exists in moving a member-function from a type A to a type B. If the types do not have any dependencies, the call-sites of the member-function of type A do not necessarily have an instance of the type B. Therefore, moving this function to this destination type requires creating a type instance at the call-site. This is illustrated in Listing 5.

```
1 /**================================================
2 Example of a missing instance for a new call-site
3 owner if a member-function is moved to another
4 type
5 ================================================**/
6 // A.h
7 struct A {
8   void foobar();
9 };
10
11 // B.h
12 struct B {
13 };
14
15 // Main.cpp
16 #include "A.h"
17
18 int main() {
19   A a;
20   // Moving to 'B', but no instance at call-site
21   a.foobar();
22 }
```

Listing (5) *Missing Class-Instance Outside-Dependency*

Of course, a similar problem exists for the visibility of type B, however, this time the type must be defined after the transformation of the call-site is finished. The same problem exists for types used by the function, for example a parameter. It must be assured they are available at the move destination.

Inside-Dependencies
• Required member-variable access no longer available after move • Required member-function access no longer available after move • Required free-function no longer visible after move.

The inside-dependencies for function based moves are heavily based on required access to fields and functions defined at the move origin. Moving a function to a new location must ensure that the used elements are still available at the destination.

Conflicts
• Overload conflict introduced by the move

C++ Language Rules
• Specific operators are only allowed as member-functions (e.g. assignment)

2.1.2 Type Based Moves

Type based move refactorings handle moving a class/struct or an enum. We will not make a separation between declaration and definition for these types of move refactorings either, for the same reason and with the same exception as described in 2.1.1 on page 6.

Refactoring	**Reverse Operation**
(1) Move Class to New/Existing File	(1) Move Class to New/Existing File
(2) Move Class to New/Existing Namespace	(2) Move Class to New/Existing Namespace
(3) Merge Class Member Declarations and Definitions	(4) Separate Class Member Declarations and Definitions
(4) Separate Class Declaration and Definition	(3) Merge Class Declaration and Definition
(5) Extract Class	(6) Inline Class
(6) Inline Class	(5) Extract Class

The move refactoring *(3) Merge Class Member Declarations and Definitions* and *(4) Separate Class Member Declarations and Definitions* are already implemented and integrated in the Eclipse *CDT* Indigo release [Sch10] as ”toggle” refactoring, but are listed for completeness.

Related Problems

Outside-Dependencies

- Types used in the class may no longer be visible at the new position
- Visibility of class no longer sufficient for call-side adjustment

These problems are similar to the outside-dependency problems described in Section 2.1.1 on page 6, but from a type point of view.

Inside-Dependencies

- Required member-variable access no longer available after move ((5) only)
- Required member-function access no longer available after move ((5) only)

These problems are similar to the outside-dependency problems described in Section 2.1.1 on page 6, but from a type point of view.

Conflicts

- Name conflict introduced by the move

C++ Language Rules

- One definition rule violations

2.1.3 Variable-Based Moves

Variable based move refactorings tackle the problem of moving global-/member-variables, global-/member-constants, static variables and typedefs. For these move refactorings, it is important to track the initialization of the variable, e.g. using the initializer list of a constructor. Therefore there may exist influences for type instantiation call-sites and initialization ordering.

Refactoring	**Reverse Operation**
(1) Move Member-Variable to another Class	(1) Move Member-Variable to another Class
(2) Move Static Member-Variable to another Class	(2) Move Static Member-Variable to another Class
(3) Move (Static) Constant Member-Variable to another Class	(3) Move (Static) Constant Member-Variable to another Class
(4) Move Global Variable to Class	(5) Move Member-Variable to Global
(5) Move Member-Variable to Global	(4) Move Global-Variable to Class
(6) Move Static Global-Variable to Class	(7) Move Static Member-Variable to Global
(7) Move Static Member-Variable to Global	(6) Move Static Global-Variable to Class
(8) Move Static (Const) Global-Variable to Class	(9) Move Static (Const) Member-Variable to Global
(9) Move Static (Const) Member-Variable to Global	(8) Move Static (Const) Global-Variable to Class
(10) Move Global-Variable to another Namespace	(10) Move Variable to another Namespace
(11) Move Local-Variable to Member-Variable	(12) Move Member-Variable to Local-Variable
(12) Move Member-Variable to Local-Variable	(11) Move Local-Variable to Member-Variable

Apparently, there are a various combinations of moving a variable from an move-source to a new location. Implementing a move refactoring for variables can become difficult rapid, because variables are the ”state foundation” of an application and interfere in the behaviour of a program. They also can have many dependencies that must be solved. The move refactorings *(11) Move Local-Variable to Member-Variable* and *(12) Move Member-Variable to Local-Variable* can be seen as a *Test-Driven Development* (*TDD*) related move, however, they represent valid move refactorings to consider for implementation.

```
/**================================================
Example of moving (const) member-variables along
with the constructor initializer-list
================================================**/
// Moving a Member-Variable before Transformation
struct Origin {
  int i;
};

struct Destination {
};

// Moving a Member-Variable after Transformation
struct Origin {
};

struct Destination {
  int i;
};

// Moving a Const Member-Variable before Transformation
struct Origin {
  Origin(int value) : i(value) {}
  const int i;
};

struct Destination {
};

// Moving a Const Member-Variable after Transformation
struct Origin {
  Origin(int value)
};

struct Destination {
  Destination(int value): i(value) {}
    const int i;
};
```

Listing (6) *Examples of Variable Based Moves*

Listing 6 shows some examples of variable based moves. However, they do not include solutions for the related problems since this task is dedicated to the design section, if an implementation is feasible in the project.

Related Problems

Outside-Dependencies

- Missing class-instances or instance-access not sufficient for variable call-side adjustment
- Visibility of classes not sufficient for variable call-side adjustment
- The type of the moved variable is not known at the new position

Inside-Dependencies

- Constant initializer may have to be moved with the constant
- A moved constant initializer from the initializer list has no exact counterpart

Conflicts

- Name conflicts introduced by the move

C++ Language Rules

- Constant-Variables must be initialized
- Reference-Variables must be initialized
- Target constructor may have to be changed
- The initialisation sequence must be retained in the target for non-static member-variables, if any

3 Move Refactorings

In Section 2.1 on page 3 we have seen that there exist various types of move refactorings. Unfortunately, it is not possible to implement all of them in this master-thesis. Therefore, this project will focus on the most valuable types of move. Each of these selected moves is analysed in detail and implemented in Eclipse *CDT*. The results are described in the following sections.

3.1 Move Member-Function to Another Class

This move refactoring type is inspired by the *"Move Method"* refactoring:

> *"A method is, or will be, using or used by more features of another class than the class on which it is defined."* [Fow04]

Instead of method the term *member-function* ([ISO11] Section 9.3) is used for this move refactoring. This is the correct term in a C++ environment and does accurately describe what we want to move.

3.1.1 Motivation

Classes naturally have assigned responsibilites. These responsibilities can be represented in different ways, but frequently the state and the behaviour is controlled and modified over the lifetime of the class-instance by using member-functions. Unfortunately, the assigned responsibilities may not be correct. The first time the code was written, it was reasonable to assign the member-function to this class, but as time proceeds and code is added or changed, the member-function suits better to another class. This can be due to *"..using more features of another class than the class on which it is defined."* [Fow04], as seen in the inspiration sentence, or our class has too much responsibilities which we preferably separate into smaller pieces to ease understanding and maintainability. Sometimes, there are also situations where we just did not understand the problem well enough to make a reasonable decision for where to put the member-function.

3.1.2 What to Move

Compared to Java, moving a member-function to another class requires more effort in C++. The traditional separation between declaration and definition requires to carefully think about what should be moved in which situations. A declaration and definition of a member-function can appear in different variations. They may be separated in two files typically a header-file for the declaration and a source-file containing the definition, but can be co-located together in the header-file or finally be located together usually found in template definitions. In addition, a definition may not bet present because an implementation was not feasible up to now, but still the need for move functionalities exist. These situations occur on both, at the move source and destination. So apparently, what to move, or what does make sense to move is a primary question for this type of move refactoring. One option is to support all types of declaration and definition arrangements. A move refactoring supporting this functionality

must allow a user to chose what to move, namely the declaration, the definition or both. In addition, the target must be examined carefully to potentially adjust the moved declaration and/or definition to match the target. For example, if the member-function declarations and definitions in the target class of our move refactoring are together, it is reasonable to move our code equally. Clearly, this adds unnecessary complexity on both, the development and the usability side. A developer using the move refactoring has to chose between a likely huge amount of move options. On the other side, implementing and testing these options is difficult and error-prone. So it is wise to introduce a superior abstraction. We are calling this a logical and a physical move of a member-function.

Introducing Logical and Physical Move Separation

The reason for a logical and a physical move separation becomes more clear if we consider the real problem tackled by this refactoring: There are faulty or misplaced responsibility assignments. Generally speaking, the discussion is about the "where" a member-function belongs to. A member-function in C++ is owned by a type, or we can say the member-function "belongs to" a type. The specification of this association is based only on the declarations. Therefore, to solve the problem, it is sufficient to move only the declaration to the new location. Compared to the first option where all combinations of declaration and definition in the source and target of the move must be considered, we separated it into two independent move refactorings, a logical move member-function refactoring, relocating the declaration and solving the assignment problem and a physical move member-function refactoring, to relocate the definition code responsible for the behaviour of the member-function.

For this type of move refactoring, we implemented the logical move to relocate the member-function declaration.

3.1.3 Applying a Move

Applying a move transformation to the declaration of a member-function does not mean we can ignore the definition. Logically relocating the declaration to another class induces changes to the declaration and potentially to the definition as well.

In the following, we will illustrate the mechanics of this move refactoring, including the transformations made, chooseable options for a user and encountered problems.

Selecting a Destination

Moving a member-function to another class requires selecting a destination class for the transformation. But which classes are good candidates for such a refactoring? Basically, the move destination is either proposed automatically or selected manually by a user. The transformation actions to apply are roughly the same, independent of whether a proposed destination type was chosen or the target was selected manually. The question is which types serve as feasible move destination recommendations. For this type of move refactoring, we used the parameters of the function to move to create move destination type proposals.

```
/**=============================================
Making proposals for a move destination if a
member-function should be moved to another type
=============================================**/
struct DstA {};
struct DstB {};
struct DstC {};

struct Source {
  // Propose move to 'DstA', 'DstB' or 'DstC'
  void functionToMove(DstA a, DstB b, DstC c);
};
```

Listing (7) *Move Destination Type Proposals*

The advantage of proposing a parameter type as a move destination is that a parameter type is likely to be an appropriate choice for the new member-function location. For example, if the function only uses functions from a parameter without changing the internal state of the own object, it is wise to move the member-function to this parameter type. In addition, it is possible to enforce a move destination proposal for a given type by just adding an extra parameter to the member-function. Moving the member-function to a manually chosen type is an additional proposal option, but with an empty destination. Thus the available proposals for this type of move is always one for the ”free move” plus a proposal for each distinct parameter type.

Changing the Declaration and Definition

As we have seen in Section 3.1.2 on page 13 the primary type of code relocation is based on the declaration. We still have the same declaration and definition arrangements, but it is possible to define the appropriate actions without requiring user interaction. So what have to be done? Here is a first simple illustration using a type *Bill*:

```
/**=============================================
Initial situation for moving the member-function
'calculatePrice' to 'Product'
=============================================**/
// Bill.h
#include "Product.h"

struct Bill {
  double getAmount() const {
    double amount = 0.0;
    for(int i=0; i<products.size(); ++i)
      amount+=calculatePrice(products[i]);
    return amount;
  }
  double calculatePrice(const Product & p) const {
    return p.getPrice() * p.getQuantity();
  }
  std::vector<Product> products;
  // ...
};
```

Listing (8) *Bill Example before Move (Together)*

The type *Bill* contains a vector of *Product*s with two member-variables for the price and the quantity bought by a customer. The price of each Product is calculated by using the member-function *calculatePrice*. However, since the product is the information expert [Lar07] for this calculation, the member-function should be moved to this type.

```
/**================================================
Result after the member-function 'calculatePrice'
was moved to 'Product'
================================================**/
// Bill.h
#include "Product.h"

struct Bill {
  double getAmount() const {
    double amount = 0.0;
    for(int i=0; i<products.size(); ++i)
      amount+=products[i].calculatePrice();
    return amount;
  }
  std::vector<Product> products;
  // ...
};

// Product.h
struct Product {
  double calculatePrice() const {
    return getPrice() * getQuantity();
  }
  // ...
};
```

Listing (9) *Bill Example after Move (Together)*

In the type *Bill* the declaration and the definition is removed and added to *Product*. Since the type of the parameter of the moved function is equal to the destination type, there is no longer a need to pass this parameter and is therefore removed. The call to *getPrice* and *getQuantity* are changed to apply on the local instance for the same reason. Last but not least, the call-site of *calculatePrice* in the member-function *getAmount* is changed to be an invocation on the instance of the product previously used as the member-function argument.

The changes made for the above examples are sufficient to ensure compilability and program semantics. But traditionally, declaration and definition are separated in a header- and a source-file. For this arrangement, more changes are necessary. To explain these transformations, the same example is used, but splitted in declarations and definitions in a header- and a source-file.

```
/**=============================================
Initial situation for moving the member-function
'calculatePrice' to 'Product' with definition change
=============================================**/
// Bill.h
#include "Product.h"

struct Bill {
  double getAmount() const;
  double calculatePrice(const Product & p) const;
  std::vector<Product> products;
  // ...
};

// Bill.cpp
#include "Bill.h"
#include "Product.h"

double Bill::getAmount() const {
  double amount = 0.0;
  for(int i=0; i<products.size(); ++i)
    amount+=calculatePrice(products[i]);
  return amount;
}

double Bill::calculatePrice(const Product & p) const {
  return p.getPrice() * p.getQuantity();
}
```

Listing (10) *Bill Example before Move (Separated)*

Here, the true power of the logical- and physical move separation becomes evident. The member-function *calculatePrice* remains in the same file, but the membership is changed to *Product*. In addition, an include statement for the product header-file is added to the source-file containing the definition. Apparently, the same procedure is applicable for co-located declarations and definitions.

```
/**=============================================
Result after the member-function 'calculatePrice'
was moved to 'Product' with changed definition
=============================================**/
// Bill.h
#include "Product.h"

struct Bill {
  double getAmount() const;
  std::vector<Product> products;
  // ...
};

// Product.h
struct Product {
  double calculatePrice() const;
  // ...
};

// Bill.cpp
#include "Bill.h"
#include "Product.h"

double Bill::getAmount() const {
  double amount = 0.0;
  for(int i=0; i<products.size(); ++i)
    amount+=calculatePrice(products[i]);
  return amount;
}

double Product::calculatePrice() const {
  return getPrice() * getQuantity();
}
```

Listing (11) *Bill Example after Move (Separated)*

After the move it is possible to get rid of the member-function calls *getPrice* and *getQuantity*, using the member-variables directly. Though this transformation task is dedicated to other refactorings such as *Inline Method* [Fow04].

In constellations where only a declaration exists, the process is even easier than in the situation where declaration and definition are together. The declaration is removed in the source type and added to the destination type, including the removal of a parameter if a parameter type is equal to the destination type. And since no definition exists, no more changes are required.

Namespaces

Namespaces are an important part of the transformation. Looking at the origin and the destination of a move, it is possible to discover different namespaces. Thus moving a member-

function from a source to a new destination potentially requires modifications to namespaces. Apparently, this problem is only present for separated or at least co-located declarations and definitions. Only in these situations definitions can have an incorrect position in the file by having a surrounding namespace. For declaration only or declaration and definition together this is not possible, because they will automatically be in the appropriate namespace due to the fact that a namespace will surround the type and cannot be defined inside the type.

To solve the problem of a namespace mismatch between the move origin and destination, we use the opportunity to use the qualified name of the destination for the function definition instead of opening and closing the namespaces required.

```
1 /**================================================
2 Initial situation for moving the member-function
3 'foo' to 'Destination' with namespace changes
4 ================================================**/
5 // Destination.h
6 namespace d_outer {
7   struct Destination {
8   };
9 }
10
11 // Source.h
12 namespace s_outer {
13   namespace s_inner {
14     struct Source {
15       void foo();
16     };
17   }
18 }
19
20 // Source.cpp
21 #include "Source.h"
22 namespace s_outer {
23   namespace s_inner {
24     // other functions before...
25     void Source::foo() {
26       // ...
27     }
28     // other functions after...
29   }
30 }
```

Listing (12) *Surrounding Namespaces before Move*

To move the member-function *foo* from *Source* to *Destination*, the same actions are applied as for a normal move, but in the file containing the definition, additional actions are required.

```
/**=============================================
Result after the member-function 'foo' was moved
to 'Destination' with changed namespaces
=============================================**/
// Destination.h
namespace d_outer {
  struct Destination {
    void foo();
  };
}

// Source.h
namespace s_outer {
  namespace s_inner {
    struct Source {
    };
  }
}

// Source.cpp
#include "Source.h"
#include "Destination.h"

namespace s_outer {
  namespace s_inner {
    // other functions before...
  }
}

void d_outer::Destination::foo() {
  // ...
}

namespace s_outer {
  namespace s_inner {
    // other functions after...
  }
}
```

Listing (13) *Surrounding Namespaces after Move*

The namespaces before the function to move are closed. This removes the function to move from the old namespace. The name of the function to move is fully-qualified with the destination type and the namespaces. After the function to move, the previously closed namespaces are reopened.

Tackling Inside Dependencies

Apparently, the bill-example used in the previous Section does not cover all possible problems that can occur. For this part, the inside dependencies introduced by the function to move are

examined.

In the bill-example, the only existing dependencies of the function to move are those related to the passed parameter *Product*. But frequently a member-function delegates work to other member-functions of the same type or is modifying member-variables. Hence after moving the function out of the type to a new location these member-function are no longer present because the function resides in another type than the member-functions invoked. We may be lucky by having equal functions at the new location, but this is rarely the case and not to mention the semantic behaviour is very likely to be different than before. Clearly, another solution for this problem is inevitable.

```
1 /**================================================
2 Initial situation for moving the member-function
3 'settle' to 'DebitCard' with parameter changes
4 ================================================**/
5 // DebitCard.h
6 struct DebitCard {
7   double getBalance() const {
8     return balance;
9   }
10  void reduce(double amount) {
11    balance-=amount;
12  }
13  double balance;
14  // ...
15 };
16
17 // Payment.h
18 #include "DebitCard.h"
19
20 struct Payment {
21  void settle(DebitCard & card) {
22    if(card.getBalance()<amount) {
23      // refuse payment...
24    }
25    card.reduce(amount);
26  }
27  double amount;
28  // ...
29 };
```

Listing (14) *DebitCard Example before Move (Together)*

In this example, a payment is made by using a debit card. The amount of the payment is checked to not exceed the balance of the card and the balance is reduced. The problem with this design is, that *Payment* requires different implementations for *settle*. Instead of using a debit card, a credit card could be used or pay by cash. Hence, we change the direction of the dependency to let the *DebitCard* settle the *Payment* as shown in Listing 15 on the next page.

```
/**================================================
Result after the member-function 'settle' was
moved to 'DebitCard' with changed parameter
================================================**/
// DebitCard.h
#include "Payment.h"

struct DebitCard {
  void settle(class Payment & newPayment) {
    if(getBalance()<newPayment.amount) {
      // refuse payment...
    }
    reduce(newPayment.amount);
  }
  double getBalance() const {
    return balance;
  }
  void reduce(double amount) {
    balance-=amount;
  }
  double balance;
  // ...
};

// Payment.h
#include "DebitCard.h"

struct Payment {
  double amount;
};
```

Listing (15) *DebitCard Example after Move (Together)*

Compared to the previous examples of Listing 8 on page 16 through Listing 11 on page 19 the function to move has an inside dependency to the member-variable *amount* which has to be satisfied. For this reason additional actions are performed. A new parameter of the source type *newPayment* is added to the moved function and accessing the member-variable *amount* is changed to take place on this new parameter. The newly added parameter is always passed by-reference for all cases. An in-parameter forward declaration using *class* is used to get rid of the include statement for the source header-file in the destination header-file. However, this only works in the arrangement where the declaration and definition are separated in a header-file and a source-file. In the above example, the forward declaration is not sufficient. Though the include statement can theoretically solve this, it introduces circular header-file dependencies, which we will not solve with this move refactoring. To tackle this problem, the includator plug-in [Fel11] can be used. Provided that all inside dependencies are const, the parameter is passed by const-reference. It is also noteworthy to mention that passing the amount itself instead of the the payment instance would be sufficient. However, this requires an in-depth analysis of the dependencies to the originating type to detect the appropriate data

to pass by parameter. This would also require an additional decision on how much should be passed directly before the complete instance is passed. A user must be able to decide which mechanism to use to pass these parameters as well, increasing the amount of configurations in the refactoring process. Therefore, always the complete type is passed as a dependency solving parameter. To overcome the dependency introduced by this, another refactoring can be used, described in Section E on page 103.

Of course this type of transformation applies to other inside dependencies. For example, member-function calls are changed equally by adding a new parameter of the source type and changing the invokation owner accordingly. The same for function invokations on member-variables where the call will be changed to a member-variable access on the parameter with the original function invokation. The only exception for these transformation are recursive calls. If a member-function call in the function to move is the function to move itself the call remains unchanged.

Unveil Visibilites

So far visibilities of inside dependencies were ignored. Unfortunately, live is not that easy and it is now time to explore this problem. In C++ there are two ways to specify visibilities, either by using labels or adding friend declarations. Visibility labels are more common, however, they allow no control over who is allowed to use the visible functionalities. Friends enable establishing a restrictive policy to control access. Nevertheless, it is mainly up to a developer to decide which option suits better, thus it should be possible to configure the type of visibility adjustment to use for a transformation.

Changing the visibility is necessary for inside dependencies introduced by used member-variables or member-funtions in the function to move. While using a private or protected inside dependency at the source type is legal, it will no longer be at the destination type.

```
/**================================================
Initial situation for moving the member-function
'settle' to 'DebitCard' with visibility changes
================================================**/
// DebitCard.h
struct DebitCard {
  double getBalance() const {
    return balance;
  }
  void reduce(double amount) {
    balance-=amount;
  }
private:
  double balance;
  // ...
};

// Payment.h
#include "DebitCard.h"

struct Payment {
  void settle(DebitCard & card) {
    if(card.getBalance()<amount) {
      // refuse payment...
    }
    card.reduce(amount);
  }
private:
  double amount;
  // ...
};
```

Listing (16) *DebitCard Example with Labels before Move (Together)*

The debit card example of Listing 14 on page 22 slightly changed by adding visibility labels illustrates this problem. The member-variables *amount* of *Payment* and *balance* of *DebitCard* are normally preceded by a private visibility, hiding internal information to the external objects. But relocating settle to *DebitCard* and changing the inside dependencies accordingly violates the newly added visibility contraints.

```
/**=============================================
Erroneous member-function 'settle' due to the
missing access to 'amount' in 'Payment'
=============================================**/
void settle(class Payment & newPayment) {
  if(getBalance()<newPayment.amount) {
    // refuse payment...
  }
  reduce(amount);
}
```

Listing (17) *DebitCard Example with Labels Visibility Violation*

We no longer have access to *amount* since the moved function no longer belongs to the same class containing this member-variable. If a user decides to use labels to change visibilities, the transformation will add additional visibility labels.

```
/**=============================================
Result after the member-function 'settle' was
moved to 'DebitCard' with changed visibility of
'amount' using labels
=============================================**/
// Payment.h
#include "DebitCard.h"

struct Payment {
private:  // (1)
public:   // (2)
  double amount;
private:  // (3)
  // ...
};
```

Listing (18) *DebitCard Example with new Labels*

The old private label visibility (1) is closed by adding a new public label (2), changing the member-variable *amount* to public. After *amount* the original visibility is restored (3) by adding an additional label with the previously closed visibility. Basically, all visibility changes using labels work this way, but there are some special cases. If multiple visibility changes are one after another, only one close, open and restore operation will be performed, to assure the minimal amount of changes. For the same reason no restore operation is performed in cases where the inside dependency to change into public is the last declaration in the source type.

```
/**=============================================
Example of changing visibilities for a sequence
of declarations in a type
=============================================**/
// Before Transformation
struct Source {
private:
  void func();
  int first;
  int second;
  int third;
};

// After Transformation
struct Source {
private:
  void func();
public: // open 'first', 'second' and 'third'...
  int first;
  int second;
  int third; // no restore after 'third'...
};
```

Listing (19) *Multiple Visibility Changes and Omitted Label Restore*

Another particular situation where a type declaration comes with multiple declarators is covered as well. In C++ it is legal to define multiple variables of the same type by using a comma-separated list of declarators. Although this is infrequently used for declaring member-variables there is existing code using this syntax. Changing the visibilities for these types of declarations must retain the order of the member-variables. Otherwise the order of initialization is changed, potentially breaking compilability due to the order of initialization given by constructors.

```
/**=============================================
Example of changing visibilities for a sequence
of declarations with multiple declarators in a type
=============================================**/
// Before Transformation
struct Source {
private:
  int x, y, z; // change 'y' to public...
};

// After Transformation
struct Source {
private:
  int x;
public:
  int y;
private:
  int z;
};
```

Listing (20) *Multiple Declarator Visibility Changes with Same Order*

The second option of using a friend modifier to change the visibility requires less changes. Following the method of minimal changes, we also rather use a friend function modifier instead of a friend type, changing only the visibility of the parts related to the move refactoring. Using the same initial situation as of Listing 16 on page 25, no labels are added, but instead a friend function.

```
/**=============================================
Result after the member-function 'settle' was
moved to 'DebitCard' with changed visibility of
'amount' using a friend function declaration
=============================================**/
// Payment.h
#include "DebitCard.h"

struct Payment {
private:
  double amount;
  friend void DebitCard::settle(class Payment &);
};
```

Listing (21) *Friend Function Visibility Modification*

The examples used for illustrating visibility changes access the member-variables after the move directly. These changes are done to ensure compilability, but we recommend to add accessor functions for the changed member-variables afterwards. However, this transformation task is dedicated to the refactoring *Encapsulate Field* [Fow04].

Adapting Call-Sites

To apply the move refactoring consistently, existing invocations of the function to move have to be changed as well. Before a call-site is adapted, several questions have to be answered. Is the move to a parameter type? Was an additional parameter added? Have we removed a parameter? These questions build the foundation of the call-site adjustments. For now, we consider a call-site of the *settle* member-function show in Listing 16 on page 25.

```
1 /**================================================
2 Initial situation for moving the member-function
3 'settle' to 'DebitCard' with call-site changes
4 ================================================**/
5 // main.cpp
6 #include "Payment.h"
7 #include "DebitCard.h"
8
9 void makePayment(DebitCard & card, Payment & payment) {
10   // log payment made by card...
11   payment.settle(card);
12   // other actions...
13 }
```

Listing (22) *Call-Site of settle before Move*

By moving *settle* from *Payment* to *DebitCard* the call-site must be changed. The function must now be invoked on the instance of *DebitCard* and not on *Payment*. For this example, our move is to a type appearing in the parameter list of the function to move, therefore, the associated argument *card* at the call-site is removed and used as invocation owner. To satisfy the inside dependencies, a new parameter of the source type is added, hence the old owner of the call-site *payment* is used as a new argument for the function call.

```
1 /**================================================
2 Result after the member-function 'settle' was
3 moved to 'DebitCard' with changed call-site
4 ================================================**/
5 // main.cpp
6 #include "Payment.h"
7 #include "DebitCard.h"
8
9 void makePayment(DebitCard & card, Payment & payment) {
10   // log payment made by card...
11   card.settle(payment);
12   // other actions...
13 }
```

Listing (23) *Call-Site of settle after Move*

Unfortunately, not all call-site changes are that easy to peform. What if no move to a parameter type is performed? In this situation, we have no instance of the new member-function owner. Hence we have to create such an instance.

```
/**=============================================
Result after a member-function was moved to a type
not in the parameter list with changed call-site
=============================================**/
// Before Transformation
void foobar(A & a) {
  // other tasks...
  a.functionToMove();
  // ...
};

// After Transformation
void foobar(A & a) {
  // other tasks...
  B().functionToMove();
  // ...
};
```

Listing (24) *Free Move Call-Site Change*

The member-function is moved to a new type B. For this, a new instance of B is created and immediatly used as the invocation owner. Thus the instance is temporary, but we achieved the goal of having minimal changes per refactoring. The situations where an immediate instantiation of the target type is not possible because the type is not default constructible are ignored in this move refactoring. A user must change these call-sites to have a legal initialization manually. After the change, the parameter a could be removed. Though this transformation task is dedicated to the refactoring *Remove Parameter* [Fow04].

We have seen that existing inside dependencies, for example by member-function calls or by accessing member-variables in the function to move, introduce a new parameter. This parameter is always passed by reference and if possible by const-reference. Unfortunately, this can lead to some problems.

```
/**================================================
Initial situation for moving the member-function
'settle' to 'DebitCard' with function evaluation
call-owner
================================================**/
// main.cpp
#include "Payment.h"
#include "DebitCard.h"

Payment getPayment() {
  // Return a received payment...
}

void makePayment(DebitCard & card) {
  getPayment().settle(card);
  // other actions...
}
```

Listing (25) *Call-Site of settle with Function Call Invocation Owner*

The invocation owner of the function to move at the call-site is the return type of another function. Although it should not be possible to invoke a member-function on such a temporary object, it actually is and is used in C++ source-code. Applying the move refactoring the same way as before will no longer work.

```
/**================================================
Erroneous call-site after the member-function
'settle' was moved to 'DebitCard' because the
payment is passed as a reference, but is temporary
================================================**/
// main.cpp
#include "Payment.h"
#include "DebitCard.h"

Payment getPayment() {
  // Return a received payment...
}

void makePayment(DebitCard & card) {
  card.settle( getPayment() );
  // other actions...
}
```

Listing (26) *Illegal Call-Site of settle with Function Call Argument*

As we have seen in Listing 15 on page 23 the new parameter is passed by reference, though the return type of *getPayment* is by value. This is not allowed, because passing a temporary object by reference is not legal in C++. Here we have to extract a local variable. This additional transformation is from the mechanical point of view similar to *Introduce Explaining*

Variable, however, the motivation is to ensure compilability, not to explain the purpose of an expression.

```
1 /**================================================
2 Result after the member-function 'settle' was moved
3 to 'DebitCard' with new local variable at call-site
4 ================================================**/
5 // main.cpp
6 #include "Payment.h"
7 #include "DebitCard.h"
8
9 Payment getPayment() {
10   // Return a received payment...
11 }
12
13 void makePayment(DebitCard & card) {
14   Payment newPayment = getPayment();
15   card.settle( newPayment );
16   // other actions...
17 }
```

Listing (27) *Legal Call-Site of settle with Function Call Argument*

Introducing a new local variable is only necessary for the above situation. If the return type of *getPayment* is by reference, we do not have to create a local variable, because passing a reference return type by reference is allowed. In addition, if the newly added parameter of the function to move is by const-reference, this is not required as well. Passing a reference by const-reference or passing a temporary object by const-reference is legal.

3.2 Move Member-Function to Another File

For this type of move refactoring, we do not have a related refactoring available. The reason for this is the traditional separation between declaration and definition in C++ that we described as a logical and a physicial membership in Section 3.1 on page 13. But the available literature about refactorings do normally not discuss such a separation. However, it is important to get a first brief impression of the problem tackled, thus we provide an own problem statement.

> *"A member-function definition in a source-file is more suitable in another source-file"*

This is not a paradigm, it should rather be seen as a helpful guidance. Apparently, it is up to a developer to decide which member-function does not belong to a given source-file or for which it may be wise to put them together. However, explicit separation from both, the logical and the physical point of view, eases understanding of source-code and maintainability. Hence, if a member-function definition owned by a type A is the only definition of a member-function of this type in a source-file, and all other member-function definitions of this type are located in another source-file, this member-function is a suitable candidate for a move.

3.2.1 Motivation

Applying the *move member-function to another class* refactoring described in Section 3.1 on page 13 introduces a new problem present for separated or co-located declarations and definitions. The transformation relocates the declaration and the definition is changed to match the new requirements introduced by the destination type, however, the physical location of the definition remains unchanged. Although this physical membership of the member-function may not be a problem, it can still be an unsatisfying situation. Assume we already have a source-file that contains member-function definitions owned by the destination type of the *move member-function to another class* refactoring. Having this member-function definition isolated from the other definitions of the same type may not be wise. In general, having member-function definitions belonging to the same type distributed over possibly many other files is probably not a desirable situation. It unnecessarily complicates understandability and maintainability of the source-code, because changing a given member-function of the type may requires looking for the definition of the member-function, which is not in the source-file together with all other definitions of this type.

Although the motivation for this type of move refactoring is heavily based on the usage of this move as a follow-up transformation for the *move member-function to another class* refactoring, it is possible to use it independently. Distributed definitions of member-functions owned by the same type over different source-files can occur whether we use the *move member-function to another class* refactoring or not. For example, other refactorings such as splitting a class by using *"Extract Class"* [Fow04] could end up with a similar situation, by moving only the declarations and retaining the physical location of the definitions.

3.2.2 What to Move

As already mentioned in the Section before, this type of move refactoring applies to member-function definitions. Although it may be reasonable to allow moving other things on a file base, we will not consider them here, because the primary scenarios for this move type are follow-up transformations for the *move member-function to another class* refactoring. We also have already established enough knowledge to perform a relocation of member-functions in a proper way.

Since we focus on moving member-function definitions, it is necessary to describe the encountered situations more precise. Theoretically, the same arrangements of declarations and definitions exist (see Section 3.1 on page 13), namely together, co-located, separated or no definition. Apparently, if no definition is present, moving a member-function definition to another file is not possible, hence this situation can be ignored. So can the situation where declaration and definition are together. Moving a member-function where the declaration and a definition appear together is completely covered by the *move member-function to another class* refactoring. Therefore, only definitions appearing in a separated or co-located arrangement are candidates for this type of move.

3.2.3 Applying a Move

For the *move member-function to another file* move refactoring, we can ignore the declaration of the member-function. Only the relocation of the definition of a member-function induce changes to the source-code.

In the following, we will illustrate the mechanics of this move refactoring type, including the transformations made, chooseable options for a user and encountered problems.

The Destination File

Like every move refactoring, a destination for the function to move is required. As the name of the refactoring implies, the move destination for this type of move is a file. The question is, which files are good candidates for the move destination? Similary to the *move member-function to another class* refactoring, it would be possible to create proposals based on the type the function to move belongs to. If the declaration of the function to move is in a header-file, we could propose all source-files with an include statement for this header-file. But unlike the *move member-function to another class* refactoring where proposals are made for each distinct parameter type plus a "free" one, this is not a good idea. The amount of source-files including a given header-file is very likely to exceed the amount of parameters of a function. Hence the amount of proposals by using this technique is also likely to be very large and selecting the appropriate target becomes more difficult for a user. In addition, using this include based approach may propose target source-files contained in other projects. Although this is not necessarily wrong, it is unlikely that these source-files are used as the move destination, because the file is only incuded to make the type available in this translation unit.

Basically, it is possible to distinguish between two base move destination proposals. Either the target is an existing file, or a new file has to be created. Moving the member-function definition to an existing file requires selecting the destination file, where moving to a new file requires specifying a name for the file to create. The follow-up transformations to apply do not differ independent of whether an existing file or a new file is used, except that using a new file obviously requires the creation of this new file.

Changing the Definition

The changes involved for the member-function definition are simple. To getting started, we assume the separated version of the *Bill* example seen in Listing 11 on page 19.

```
/**=============================================
Initial situation for moving the member-function
definition of 'calculatePrice' to 'Product.cpp'
=============================================**/
// Bill.h
#include "Product.h"

struct Bill {
  double getAmount();
  std::vector<Product> products;
  // ...
};

// Product.h
struct Product {
  double calculatePrice();
  // ...
};

// Bill.cpp
#include "Bill.h"
#include "Product.h"

double Bill::getAmount() {
  double amount = 0.0;
  for(int i=0; i<products.size(); ++i)
    amount+=calculatePrice(products[i]);
  return amount;
}

double Product::calculatePrice() {
  return getPrice() * getQuantity();
}
```

Listing (28) *Bill Example with Definitions of Different Types before Move*

The file *Bill.cpp* contains two member-function definitions, *getAmount* of *Bill* and *calculatePrice* of *Product*. The location of *getAmount* seems fine, because it is a member of *Bill* and the containing file is *Bill.cpp*. However, *calculatePrice* is a member of *Product*, hence we may wish to relocate this member-function definition to the file *Product.cpp*.

```
/**=============================================
Result after the member-function definition of
'calculatePrice' was moved to 'Product.cpp'
=============================================**/
// Bill.cpp
#include "Bill.h"
#include "Product.h"

double Bill::getAmount() {
  double amount = 0.0;
  for(int i=0; i<products.size(); ++i)
    amount+=calculatePrice(products[i]);
  return amount;
}
// ...

// Product.cpp
#include "Bill.h"
#include "Product.h"

double Product::calculatePrice() {
  return getPrice() * getQuantity();
}
// ...
```

Listing (29) *Bill Example with Definitions of Different Types after Move*

The definition is removed from the file *Bill.cpp* and added to the file *Product.cpp*. In addition, the includes present at the move source are added to the destination file as well. This ensures compilability, because this way everything required by the moved function is also available in the destination file. Existing include statements of the destination file have to be considered as well to avoid adding an already existing include statement to the destination file.

Adding Namespace Name-Qualifiers

Relocating a member-function definition to another file requires to have a look at surrounding namespaces. Although it is possible to already have namespace names in the qualified name of the definition, we have to make sure that all namespaces are included in the definition at the move target. To satisfy this, we use the same mechanism as described in Section 3.1 on page 13, by looking up all surrounding namespaces of the function definition and adding them to the name of the function definition at the destination file.

```
/**=============================================
Example of changing the owning namespaces of
the moved member-function to be fully-qualified in
the definition name instead of surrounding namespace
=============================================**/
// Definition in the Origin x.cpp

namespace s_outer {
  namespace s_inner {
    void Source::functionToMove() {
      // ...
    }
  }
}

// Definition in the Destination y.cpp

// other declarations and definitions before...
void s_outer::s_inner::Source::functionToMove() {
  // ...
}
```

Listing (30) *Adding Namespace Names to the Definition*

To avoid problems problems with existing namespaces in the destination file, the definition of the function to move is added as the last element in the destination file. Hence the definition will not be contained in a surrounding namespace, however, if the same namespaces are available in the destination file the definition will not be inserted in this namespace either.

3.3 Move Member-Function to Free-Function

This refactoring is not one of the traditional move refactorings, since it is rather related to C++ or other programming languages allowing functions to be out of a class scope. However, in C++ it is normal to have such free-functions to introduce operators, helper factory functions such as *std::make_pair* and in the context of static polymorphism using templates. Especially if the considered function does not change or require direct access to the state of the object it is common to have it as a free-function. For getting started, we introduce our problem statement for this refactoring.

> *"A member-function of a type does not require access to the internal representation of the object."*

Apparently, there is another reason to have a free-function instead of a member-function, propably more often used with operators. In cases where we wish to implement an operator function for an existing type we cannot change, a free-function is the only available option. However, this use case is not considered here, since this move refactoring tackles the transformation of an existing member-function into a free-function. For a type we cannot change (e.g. from libraries), we can neither add nor remove member-functions, hence, this refactoring cannot be applied.

3.3.1 Motivation

Traditionally, member-functions of a type are used to control the behaviour and the lifecycle of an object instance. To do this, they access the internal state of the object, potentially altering this state. These member-functions require access to the internal representation of the object to do their job. However, what if a member-function does not require access to the internal representation of an object instance? They can access the internal state, but they do not need to, thus could do more than they actually should. Such a member-function should rather be implemented as a free-function. Transforming this type of member-function to a free-function retains the semantic of the code, but we achieve a lower coupling by introducing this separation. By converting a member-function to a free-function, we can also use the power of C++ static polymorphism using templates, rather than overriding virtual functions and adding unnecessary call-time overhead for the function lookup in the virtual table at runtime. An implementation of a transformation from an explicitly typed function into a template function was implemented for Eclipse in [Thr10].

3.3.2 What to Move

The transformation of a member-function into a free-function suffers from the same problems as the *move member-function to another class* refactoring. The arrangments of declaration and definition can yield numerous options on how the move should be performed. However, since we already introduced a solution for this, namely the separation between a logical and a physical relocation of declarations and definitions, we apply this here as well. The member-function declaration is relocated to a new destination becoming a free-function and the definition is changed reflecting the changes as well to ensure compilability. This correlates with the logical move transformation seen in the *move member-function to another class* refactoring.

3.3.3 Applying a Move

In the following sections, we will explain the required changes to transform a member-function into a free-function, choosable options for a user and problems that can occur.

The Move Destination

Similar to the other moves explained in Section 3.1 on page 13 and 3.2 on page 32, this refactoring requires a destination for the function that should be moved. Which destinations are good candidates for this refactoring? Basically, we could move the function declaration to a different file or keep it in the same file. But moving the function out of the current file introduces some problems, for example, the target file can have namespaces that are likely to be different than those in the originating file. Making sure the include order and the required visibility of declarations and definitions is sufficient becomes harder as well. In addition, the problem tackled by this refactoring is not a wrong responsibility assignment. It is a more subtle distinction between required and actual access to the internal state of an object. Thus, the responsibility assignment to the type is not wrong, but coupling can be reduced by converting it into a free-function. So retaining the function declaration in the

same file is the appropriate choice. This also has the advantage that only one proposal for the move destination exists, making it easier for a user to handle and by placing the function below the originating type, it is guaranteed that the moved function has all the required types and function visibilities.

Changing the Declaration and Definition

To illustrate the required changes to the declaration and the definition of the function to move, we use operators related to a type *Fract*.

```
/**================================================
Initial situation for converting the member-operator
'operator*' into a free-operator
================================================**/
// Fract.h
struct Fract {
  Fract operator*=(const Fract & rhs) {
    numerator*=rhs.numerator;
    denominator*=rhs.denominator;
    // cancel fraction...
    return *this;
  }
  Fract operator*(const Fract & rhs) const {
    Fract lhs(*this);
    lhs*=rhs;
    return lhs;
  }
private:
  int numerator;
  int denominator;
};
```

Listing (31) *Dependent Operators for Fraction before Move (Together)*

The multiplication operator is implemented in terms of the multiplication-assignment operator. Since the multiplication-assignment operator is public, access to the internal state of the object is not required by the multiplication operator. Hence, this member-operator can be transformed into a free-operator.

```
/**================================================
Result after the member-operator 'operator*' was
converted into a free-operator
================================================**/
// Fract.h
struct Fract {
  Fraction operator*=(const Fract & rhs) {
    numerator*=rhs.numerator;
    denominator*=rhs.denominator;
    // cancel fraction...
    return *this;
  }
private:
  int numerator;
  int denominator;
};

Fract operator*(const Fract & lhs, const Fract & rhs) {
  Fract fract(lhs);
  fract*=rhs;
  return fract;
}
```

Listing (32) *Dependent Operators for Fraction after Move (Together)*

The declaration and definition of the operator, is removed and added below *Fract*. An additional parameter is added to reflect the previously implicit parameter *Fract*. This newly added parameter must be inserted as the first parameter. Otherwise it may be possible that either the operator semantic is changed, because the parameter order is changed, or the operator "owner" changes if the previous right-hand-side of the operator is not of the type *Fract*.

These changes are sufficient so far for the situation where declaration and definition are together. But consider the arrangment where declaration and definition are separated, potentially in distinct header- and source-files.

```
/**================================================
Initial situation for converting the member-operator
'operator*' into a free-operator with definition change
================================================**/
// Fract.h
struct Fract {
  Fract operator*=(const Fract & rhs);
  Fract operator*(const Fract & rhs) const;
private:
  int numerator;
  int denominator;
};

// Fract.cpp
#include "Fract.h"

Fract Fract::operator*=(const Fract & rhs) {
  numerator*=rhs.numerator;
  denominator*=rhs.denominator;
  // cancel fraction...
  return *this;
}

Fract Fract::operator*(const Fract & rhs) const {
  Fract lhs(*this);
  lhs*=rhs;
  return lhs;
}
```

Listing (33) *Dependent Operators for Fraction before Move (Separated)*

Apparently, to transform the member-operator in a free-operator, the name of the definition has to be changed to no longer contain the type qualifier.

```
/**=============================================
Result after the member-operator 'operator*' into
a free-operator with changed definition
=============================================**/
// Fract.h
struct Fract {
  Fract operator*=(const Fract & rhs);
private:
  int numerator;
  int denominator;
};

Fract operator*(const Frac & lhs, const Fract & rhs);

// Fract.cpp
#include "Fract.h"

Fract Fract::operator*=(const Fract & rhs) {
  numerator*=rhs.numerator;
  denominator*=rhs.denominator;
  // cancel fraction...
  return *this;
}

Fract operator*(const Frac & lhs, const Fract & rhs) {
  Fract fract(lhs);
  fract*=rhs;
  return fract;
}
```

Listing (34) *Dependent Operators for Fraction after Move (Separated)*

The same transformation applies if the declaration and definition are co-located. In cases where no definition exists, moving the declaration is still possible as seen in Listing 31 on page 39 and 32 on page 40, but without a need to apply transformations related to the definition.

It is important to say that the transformation of an operator from a member- into a free-function always requires adding an additional parameter. Otherwise, the arity of the operator is changed, which is not allowed for C++ operators [ISO11]. There exist operators not allowed as free-function as well [ISO11], i.e. assignment, new, delete, subscript and the function-call operator. Although the transformation is possible, it suffers from losing compilability. Non-operator member-functions may have an additional parameter, depending on whether dependencies to the source typ exist. If there are none, no new parameter will be added.

Namespaces

Transforming a member-function into a free-function requires to have a look at the namespaces, however, nearly no changes are involved to apply the refactoring in a proper way. As we

already have discussed, the problem tackled by this refactoring is not a wrong responsibility assignment. It is the problem of having access to the internal state of an object, but it is not required. This means, that the considered function's logical membership should remain unchanged, thus have a parameter for the original type if required or always in case of operators, and retain the namespaces of the move origin. To illustrate this, consider the example of Listing 35.

```
1 /**================================================
2 Example of converting a member-function into a
3 free-function and retaining the namespace member-
4 ship
5 ================================================**/
6 // A.h before transformation
7 namespace NS_I {
8   struct A {
9     void foobar()
10    {
11      // do something...
12    }
13  };
14 }
15
16 // A.h after transformation
17 namespace NS_I {
18   struct A {
19   };
20
21   void foobar()
22   {
23     // do something...
24   }
25 }
```

Listing (35) *Retain Namespaces at Destination (Together)*

The member-function *foobar* is transformed into a free-function (for brevity, we assume that no inside- and outside-dependencies exist). The member-function is removed and added below the type A in the same namespace. In this arrangment, where declaration and definition are together, it is just necessary to make sure the free-function is added to the same namespace containing A, because it is not possible to open another namespace in the type A.

In cases where the definition and the declaration are separated, an additional transformation is required. For the declaration, the same procedure as described above is sufficient. But the name of the function in the definition must reflect the correct namespace after the change as well. Basically, there are two distinct situations possible, but with the same solution. Either the definition has a surrounding namespace, or the namespaces are present in the function definition name itself by using a qualified name (it is possible to have them combined, however,

the transformations are the same). In both cases, the type qualifier is removed from the definition to convert the member-function into a free-function.

```
/**================================================
Example of converting a member-function into a
free-function and retaining the namespace member-
ship with definition changes
================================================**/
// A.h
namespace NS_I {
  struct A {
    void foo();
    void bar();
  };
}

// A.cpp before transformation
namespace NS_I {
  void A::foo()
  {
  }
}

void NS_I::A::bar()
{
}

// A.cpp after transformation
namespace NS_I {
  void foo()
  {
  }
}

void NS_I::bar()
{
}
```

Listing (36) *Retain Namespaces at Destination (Separated)*

The same changes apply for co-located declarations and definitions. If no definition exists, the declaration changes described in Listing 35 on the previous page are sufficient.

Changing Call-Sites

The call-site adaption for the *move member-function to free-function* refactoring partially suffers from the same problems as the *move member-function to another class* refactoring. This gives us the opportunity to use the same approaches to solve these problems. But it is important to distinguish between the conversion of a non-operator member-function to a

free-function and the transformation of an operator function, or more specific, a separation between operator-syntax and function-syntax at the call-site.

```
/**================================================
Examples of different syntaxes to use operators
================================================**/
// Operator-Syntax at call-site
Type x = y * z;

// Function-Syntax at call-site
Type x = y.operator*(z);
```

Listing (37) *Operator- and Function-Syntax at Call-Site*

Using an operator with function-syntax can occur by renaming a previously non-operator to an operator function, or to avoid ambiguity. Apparently, if the operator-syntax is used, the call-site remains unchanged. There is no syntactical difference between a free-function operator and a member-function operator if used in the ”traditional” way of operator invocation. However, if the operator is invoked using the function-syntax the call-sites must be changed and of course the same applies to every other non-operator function call-site as well, since ”normal” functions are always invoked using the function-syntax. To illustrate these changes, the same example with the type *Fract* seen in Listing 31 on page 39 is used.

```
/**================================================
Initial situation for converting the member-operator
'operator*' into a free-operator with function-syntax
call-site
================================================**/
// Fract.h
struct Fract {
  // Constructor with initialization...
  Fract operator*=(const Fract & rhs) {
    numerator*=rhs.numerator;
    denominator*=rhs.denominator;
    // cancel fraction...
    return *this;
  }
  Fract operator*(const Fract & rhs) const {
    Fract lhs(*this);
    lhs*=rhs;
    return lhs;
  }
private:
  int numerator;
  int denominator;
};

// Main.cpp
#include "Fract.h"

int main() {
  Fract f1(1, 2);
  Fract f2(3, 4);
  Fract result = f1.operator*(f2);
}
```

Listing (38) *Adapting Call-Sites to Free-Functions before Move*

The transformations required for this type of call-site are similar to those of the *move member-function to another class* refactoring, but without having a new owner for the function to move.

```
1 /**================================================
2 Result after the member-operator 'operator*' was
3 converted into a free-operator with function-syntax
4 call-site
5 ================================================**/
6 // Main.cpp
7 #include "Fract.h"
8
9 int main() {
10   Fract f1(1, 2);
11   Fract f2(3, 4);
12   Fract result = operator*(f1, f2);
13 }
```

Listing (39) *Adapting Call-Sites to Free-Functions after Move*

The original owner of the function invocation *f1* is removed and no new owner is added, because the function is now a free-function. Instead, *f1* is added as a new first argument to the function call. For operators, this add operation is mandatory and must not be omitted, even if no inside-dependencies exist, however, adding the original invocation owner may be skipped for non-operator functions if no inside-dependencies exist.

Similar to the *move member-function to another class* refactoring, it is possible to be forced to introduce a new local variable to ensure compilability. If the original invocation owner is itself a function call and the return-type of this function call violates the parameter passing mechanism, a new local variable must be created.

```
1 /**================================================
2 Example of erroneous and correct call-site change
3 after the conversion of a member-operator into
4 a free-operator because of a temporary reference
5 parameter
6 ================================================**/
7 // Main.cpp
8 #include "Fract.h"
9
10 Fract getFraction()
11 {
12   return Fract(1, 2);
13 }
14
15 // (1) Not allowed
16 int main() {
17   Fract f(3, 4);
18   Fract result = operator*( getFraction() , f2);
19 }
20
21 // (2) Correct
22 int main() {
23   Fract f(3, 4);
24   Fract newFract = getFraction();
25   Fract result = operator*( newFract , f2);
26 }
```

Listing (40) *Call-Site of Free-Functions with new Variable*

(1) is not allowed, because the new argument is the by-value return-type of a function passed by-reference to the operator, but passing a temporary object by-reference is not legal in C++. Thus, in (2) a new variable is added initialized with the return value of *getFraction*. This new variable can be passed to the function without a compilation error.

Adding Forward Declarations

So far, we did not have to deal with adding forward declarations for the function to move. In the *move member-function to another class* refactoring, this problem is solved by having include statements making sure everything is visible at the point it is used (except the potential circular dependencies). The *move member-function to another file* refactoring does not have this problem either, because of the newly added include statements as well. Unfortunately, this refactoring can encounter a situation, where it is unavoidable to add a forward declaration for the moved function.

```
/**================================================
Example of a member-function dependency-chain
causing problems to a conversion into a free-function
================================================**/
// A.h
struct A {
  void x() {
  }
  void y() {
    x();
  }
  void z() {
    y();
  }
};
```

Listing (41) *Dependency Chain of Functions before Move*

The problem comes down to having a "dependency chain". In the example above, a chain z → y → x exists. The order in which the functions are declared is not relevant for types, however, if the function y should be converted into a free-function, we cannot avoid to add a forward declaration for y before the call-site of y in z. Normally, the refactoring is adding the function below the origin type. However, by doing so the function z does not know about y. This yields a compilation error. Unfortunately, we neither can put the function above the type, because y has a dependency to the type A due to the invocation of x. A forward declaration of A would not solve this problem, because the function is implemented inline. Therefore, the function to move is added below the type and a forward declaration is added to the call-site of y.

```
/**================================================
Example of a solved member-function dependency-chain
by adding forward declarations
================================================**/
// A.h
struct A {
  void x() {
  }
  void z() {
    void y(const class A &);
    y(*this);
  }
};

void y(const class A & newA) {
  newA.x();
}
```

Listing (42) *Dependency Chain of Functions after Move*

If multiple dependencies to y exist, only one forward declaration is added, but before the type. This introduces less changes to the source-code compared to adding a forward declaration for each call-site.

Adding forward declarations is not necessary if the process of changing visibilities introduced a function friend declaration to the originating type. The friend declaration serve the same purpose in this context like a forward declaration.

Visibilities and Inside Dependencies

To change the visibilities required by the function to move and the adjustment of the inside dependencies to the newly added parameter, we use the same mechanisms as explained in Section 3.1 on page 13. The only difference is that for the visibilities a friend declaration is preferred over adding labels. Thus, this is the default behaviour.

3.4 Move-Up to Parent Namespace

Separating code into namespaces is a common approach to group parts of your application belonging together into logical pieces. While this adds another level of abstraction to ease understanding and managing code, it can also introduce familiar assignment and dependency problems. The question which namespace a function or type should be assigned to can become difficult. Especially, if dependencies to other namespaces are considered, maybe depending on whether they are in the same namespace hierarchy, below, above, or not in the same hierarchy at all. Conflicting names and overloads must be taken care of as well. While the code base of an application changes and evolves over time, it requires a regular reassessment of these "owner assignments", otherwise you risk having bad quality code, which is hard to maintain and change.

> *"You have code in a namespace which is not the optimal choice to be the owner of this code."*

We will not discuss how to evaluate the optimal namespace in this thesis. We also do not want to give a wrong impression of namespaces. They are not meant to be a "design tool". Grouping code that belongs together is reasonable, but extensive usage of namespaces including multiple hierarchy levels is not a good concept for code separation in C++.

3.4.1 Motivation

As an application grows, it is important to think about namespaces. Although it requires less brain-work having everything in the global namespace, it can become cumbersome very fast. For example, if there are many free-functions in your application code, the auto-completion of the *IDE*'s editor loses it purpose, because you have to chose the correct function out of a huge list of available candidates. In addition, the readability of your code suffers from the missing grouping of elements, since namespace names give you extra information about the invoked function or type used. As a project advances and the code is reassessed, you may find a function or a type which is in an incorrect namespace. It may have been placed in a namespace which was wrong in the first place, because you thought it is wise to put it there.

Or a given set of functions or types should be moved to a separate namespace to allow easily replacing them via a using namespace statement at the usage scope. Doing such a move manually is tedious and error prone and should be available in an automated fashion.

3.4.2 What to Move

Moving code between different namespaces requires thinking carefully about the way it should be performed. A relocation from an originating to a target namespace, for which both are in distinct namespace hierarchies is difficult, because the dependencies must be changed to relate on a completely different hierarchy. In addition, many changes to the code are required and managing the source-code becomes hard if you retain the moved function or type in the originating file (physically). It is possible to allow a user to chose a target file next to the target namespace, however, this yields even more changes. While this is not a bad thing from a technical point of view, it is for a software engineer using the refactoring. The more changes required, the harder it is to understand the result. Keeping changes as small as possible allows a user to maintain a picture of the result of the transformation in mind and simplifies answering the question whether this is the desired change to the source-code. Moving a piece of code to a namespace in a distinct hierarchy is also less likely to be an existing use-case. In fact, it is probably required to relocate source-code in the current namespace hierarchy, for example because this part of the application received an improved abstraction. For this reason, we introduce a ”Move-Up” and ”Move-Down” refactoring for namespaces.

Introducing Move-Up and Move-Down in Namespaces

Instead of manually selecting a namespace as a move destination, the current namespace hierarchy is used to determine the target namespace. Depending on the selected direction, the affected code is relocated to the parent namespace, or to a child namespace. The advantage of this separation is that the required changes are small and easier to understand. In addition, it is still possible to move a given piece of code to a different namespace hierarchy by successively moving it up to the global namespace and chose a different hierarchy for the move down operation.

In this thesis, only the move-up operation is implemented because there was not enough time available to complete both refactoring directions. It is also noteworthy that only free-functions and types are considered to be valid candidates for the move-up refactoring. It is possible to add global constants or variables and other C++ elements in the future, but due to time constraints, we focused on types and free-functions. This also integrates well into the previous refactorings described in Section 3.1 on page 13, 3.2 on page 32 and 3.3 on page 37, where member-functions and free-functions are handled.

3.4.3 Applying a Move

In the following sections, we will explain the required changes to move-up a free-function or a type to the parent namespace, choosable options for a user and problems that can occur.

The Move Destination

Apparently, the destination of the ”move-up to parent namespace” is very easy to point at. A namespace can only have one parent. Therefore, discussing appropriate move proposals is unnecessary. However, the only important thing to point out here is the global namespace. While the global namespace can be a valid move destination, it can never occur as a move origin in the move-up process. This is due to the fact that no parent namespace exists for the global namespace.

Changing Namespace Ownership

Changing the namespace ownership of a free-function or a type is a problem that sounds familiar. In the *move member-function to another class* refactoring in Section 3.1 on page 13, a mechanism was introduced to change the definition of a member-function to the namespace the target type for the move belongs to. This technique closes all existing namespaces before the considered function, changes the name of the member-function definition to be fully-qualified and reopens the previously closed namespaces after this definition. The algorithm can be adapted for the move-up transformation.

```
1 /**================================================
2 Initial situation for moving 'mean' up to the parent
3 namespace
4 ================================================**/
5 // MathUtility.h
6 namespace util {
7   // other functions before...
8   double mean(const std::vector<double> & v);
9   // other functions after...
10 }
```

Listing (43) *Moving-Up in a Namespace before Transformation*

Listing 43 shows an initial situation for the *move-up to parent namespace* refactoring. The free-function *mean* has to be moved out of the namespace *util*.

```
1 /**================================================
2 Result after 'mean' was moved-up to the parent
3 namespace
4 ================================================**/
5 // MathUtility.h
6 namespace util {
7   // other functions before...
8 }
9 double mean(const std::vector<double> & v);
10 namespace util {
11   // other functions after...
12 }
```

Listing (44) *Moving-Up in a Namespace after Transformation*

The transformations applied are exactly the same as the changes related to namespaces in the *move member-function to another class* refactoring. However, this is only because the move target is the global namespace. In cases where the move origin is a nested namespace, only the originating namespace is closed, moving the free-function one level up. In addition, no changes to the name of the free-function are applied, independent of whether it is a definition or a declaration. These changes apply for all declaration and definition arrangements. For separated and co-located declarations and definitions the transformation is applied for both independently. If there is no separate declaration only the definition is changed.

Inside Dependencies and *Argument Dependent Lookup*

Moving a free-function or a type to the parent namespace introduces changes to the inside dependencies. The moved code may use functions or types defined in the originating namespace. Thus, it is necessary to add qualifiers or using declarations to these dependencies to ensure retained semantics.

The transformations of the inside dependencies are performed with respect to *Argument Dependent Lookup* (*ADL*). With this technique, some of the inside dependencies do not require changes. In the context of the move-up to parent namespace refactoring, every free-function invoked in the type or free-function to move does not require a qualifier or using declaration, if the function can be looked up using *ADL*. A lookup is successful if the invoked function has at least one non-primitive parameter type and the definition of the type is in the same namespace as the invoked function. Of course, this only applies to unqualified calls. Qualified calls are looked up in the defined lookup namespace of the call. To illustrate this, assume the example of Listing 45 on the next page.

```
/**=================================================
Example of argument dependent lookup used at the
call-site of functions
=================================================**/
// ADL.h - Successful ADL
namespace A {
  namespace B {
    struct X {};
    void bar(X x) {}
  }
  void foo() {
    B::X x;
    bar(x); // (1) ADL Successful
  }
}

// ADL.h - Not Successful ADL
namespace A {
  struct X {};
  namespace B {
    void bar(X x) {}
  }
  void foo() {
    X x;
    bar(x); // (2) ADL Not Successful
  }
}
```

Listing (45) *Argument Dependent Lookup*

The invocation (1) is successful, because the local variable of type X adds the namespace B to the search scope for the function *bar*. However, (2) is not successful and will not compile, because the lookup of *bar* will not search in namespace B. For the invocation (2), the refactoring is adding a qualifier for the owning namespace, illustrated in Listing 46 on the following page. The same applies for types used in the function or type to move-up.

```
/**================================================
Result of a corrected unsuccessful argument dependent
lookup
================================================**/
namespace A {
  struct X {};
  namespace B {
    void bar(X x) {}
  }
  void foo() {
    X x;
    B::bar(x);
  }
}
```

Listing (46) *Transformed Inside Dependency with Qualifier*

In some cases, it is not possible to add a qualifier to the function invocation. Namely, these are operators implemented as free-functions and used with the traditional operator-syntax. Member-Function operators do not belong to this category because they are automatically introduced in the lookup scope by the operator argument types and their owning namespaces. To solve this problem, it is possible to add a using declaration introducing the qualified name of the operator to be used before the inside dependency.

```
/**================================================
Initial situation for a lookup correction in the
move-up to parent namespace tansformation
================================================**/
struct Fract {
  // ...
};
namespace fraction_math {
  Fract operator/(int num, const Fract & den) {
    // ...
  }
  Fract reciprocal(const Fract & f) {
    Fract inv = 1 / f;
    return inv;
  }
}
```

Listing (47) *Using Declaration for Operators before Transformation*

Listing 47 shows an initial situation for this transformation. The free-function *reciprocal* is intended to be moved-up, but since the type *Fract* and the division operator are not defined in the same namespace, *ADL* is not sufficient.

```
1 /**================================================
2 Result after the move-up of 'reciprocal' to the
3 parent namespace with using declaration to allow
4 using 'operator/'
5 ================================================**/
6 struct Fract {
7   // ...
8 };
9 namespace fraction_math {
10   Fract operator/(int num, const Fract & den) {
11     // ...
12   }
13 }
14 Fract reciprocal(const Fract & f) {
15   using fraction_math::operator/;
16   Fract inv = 1 / f;
17   return inv;
18 }
```

Listing (48) *Using Declaration for Operators after Transformation*

The using declaration added before the operator call-site introduces the name and the defining namespace for this operator to the moved-up function to ensure compilability. If a given operator is used multiple times in the same function, only one using declaration is added before the first occurrence. Although it is possible to add the using declarations at the beginning of the function, doing so breaks the ”mental connection” between the operator call-site and the using declaration. It is harder to recognise the reason for the using declaration, if it is the first statement in a function, but the introduced name is used in the ending part of the function. For the move-up operation of a type, using declarations and qualifiers are added as well, but for each member-function. Unfortunately, this can involve having repeated using declarations distributed over different members of the type. A using declaration at class scope could solve this problem, however, class scope using declarations are only allowed to introduce names found in the inheritance hierarchy of this type ([ISO11] Section 7.3.3 Paragraph 3). Adding the using declaration before the type at the namespace level should be avoided as well, since it will introduce the name to every declaration and definition in the namespace after the type to move-up.

Changing the inside dependencies of a free-function or type to move-up to parent namespace must deal with the return and parameter types as well. If they are defined in the originating namespace, the originating namespace is added as a qualifier to these types.

Call-Site Transformation

The relocation of a free-function or a type to the parent namespace can require changes to the call-sites of the free-function or the usage-sites of the type. Next to the well-known function invocations and type instantiation call-site types, declaration expressions, such as typedefs, function return types and parameters of the moved type and using declarations must be

considered as well. The transformation also considers *ADL*, but in a different context. This time, the transformation can break a previously successful lookup. Fortunately, it is possible to sort out a subset of the call-sites, which will remain unchanged. In nested namespaces, everything defined or declared in surrounding namespaces is available to inner namespaces. Therefore, call-sites in the originating namespace of the function or type to move can remain unchanged, because they are still accessible to them. Neither will there be a problem with declaration order, since the declaration order remains unchanged by the refactoring.

```
1  /**================================================
2  Initial situation for the move-up of 'reciprocal'
3  to the parent namespace with call-site changes
4  ================================================**/
5  // Fraction.h
6  struct Fract {
7    // ...
8  };
9  namespace fraction_math {
10   Fract operator/(int num, const Fract & den) {
11     // ...
12   }
13   Fract reciprocal(const Fract & f) {
14     Fract inv = 1 / f;
15     return inv;
16   }
17 }
18
19 // Main.cpp
20 #include "Fraction.h"
21
22 int main() {
23   Fract f(1, 5);
24   Fract result = fraction_math::reciprocal(f);
25   // ...
26   return 0;
27 }
```

Listing (49) *Call-Site in Move-Up before Transformation*

Listing 49 illustrates an initial situation for a call-site transformation. The free-function *reciprocal* is moved to the parent namespace. Therefore, the invocation using the namespace qualifier in *Main.cpp* becomes invalid.

```
/**=============================================
Result after the move-up of 'reciprocal' to the
parent namespace with changed call-site
=============================================**/
// Fraction.h
struct Fract {
  // ...
};
namespace fraction_math {
  Fract operator/(int num, const Fract & den) {
    // ...
  }
}
Fract reciprocal(const Fract & f) {
  using fraction_math::operator/;
  Fract inv = 1 / f;
  return inv;
}

// Main.cpp
#include "Fraction.h"

int main() {
  Fract f(1, 5);
  Fract result = ::reciprocal(f);
  // ...
  return 0;
}
```

Listing (50) *Call-Site in Move-Up after Transformation*

The call-site is changed by removing the qualifier corresponding to the originating namespace of the moved free-function. For this situation, this is sufficient to retain compilablity.

Changing the example of Listing 49 on the previous page to use *ADL*, the move-up operation can break the previously successful lookup.

```
/**================================================
Initial situation for the move-up of 'Fract'
to the parent namespace with call-site using
argument dependent lookup
================================================**/
// Fraction.h
namespace fraction_math {
  struct Fract {
    // ...
  };
  Fract operator/(int num, const Fract & den) {
    // ...
  }
  Fract reciprocal(const Fract & f) {
    Fract inv = 1 / f;
    return inv;
  }
}

// Main.cpp
#include "Fraction.h"

int main() {
  fraction_math::Fract f(1, 5);
  fraction_math::Fract result = reciprocal(f);
  // ...
  return 0;
}
```

Listing (51) *Call-Site with ADL before Transformation*

The type *Fract* adds the namespace *fraction_math* to the lookup scope for *reciprocal*. However, moving this type to the parent namespace will break *ADL*, because *Fract* and *reciprocal* are no longer in the same namespace, thus the lookup fails. Therefore, qualifiers are added to this ”indirect dependency”.

```
/**================================================
Result after the move-up of 'Fract' to the parent
namespace with changed call-site because argument
dependent lookup is broken
================================================**/
// Fraction.h
struct Fract {
  // ...
};
namespace fraction_math {
  Fract operator/(int num, const Fract & den) {
    // ...
  }
  Fract reciprocal(const Fract & f) {
    Fract inv = 1 / f;
    return inv;
  }
}

// Main.cpp
#include "Fraction.h"

int main() {
  Fract f(1, 5);
  Fract result = fraction_math::reciprocal(f);
  // ...
  return 0;
}
```

Listing (52) *Call-Site with Broken ADL after Transformation*

If *ADL* breaks because of the transformation and the dependency is an operator used with operator-syntax, a using declaration is added instead of qualifiers.

In the *move member-function to another class refactoring* in Section 3.1 on page 13, an option is described to avoid adjusting the call-sites by retaining the original member-function, and delegating to the new member-function. For this refactoring, a similar technique can be used. The delegation mechanism adds a using declaration at the namespace level reintroducing the name of the moved free-function or type to the originating namespace.

```
/**================================================
Initial situation for the move-up of 'mean' to the
parent namespace avoiding call-site adjustments
================================================**/
// Utility.h
namespace util {
  // before...
  double mean(const std::vector<double> & v);
  // after...
}

// Main.cpp
#include <vector>
#include "Utility.h"

int main() {
  std::vector<double> v;
  // add values to 'v'...
  double m = util::mean(v);
  // ...
  return 0;
}
```

Listing (53) *Avoiding Call-Site Changes before Transformation*

In Listing 53, the call-site of *mean* in *Main.cpp* should remain unchanged, thus a user can chose to add a using declaration to the originating namespace.

```
/**================================================
Result after the move-up of 'mean' to the parent
namespace with avoided call-site adjustments by
adding a using declaration
================================================**/
// Utility.h
namespace util {
  // before...
}
double mean(const std::vector<double> & v);
namespace util {
  using ::mean;
  // after...
}

// Main.cpp
#include <vector>
#include "Utility.h"

int main() {
  std::vector<double> v;
  // add values to 'v'...
  double m = util::mean(v);
  // ...
  return 0;
}
```

Listing (54) *Avoiding Call-Site Changes after Transformation*

This option can be used for every move-up operation, whether a type, operator or a free-function is used. It is noteworthy to mention that using declarations can be used in chain. The move-up to parent namespace refactoring can be applied multiple times to a free-function or type until the global namespace is reached. If a using declaration is added in every of this operation, the first using declaration will point to the second, travelling up the hierarchy until the last using declaration, which points to the moved function or type. However, if the option to add a using declaration is not chosen by a user, or if there are existing using declarations, they are changed according to the move operation by changing the qualified name of the using declaration.

4 Implementation

In this chapter, an overview of the most important implementation concepts is given, including important design decisions. First, implementation details for all refactorings are described, explaining the creation of the move proposals and data calculation, followed by a description of implementation facts related to a specific refactoring.

4.1 Eclipse Plug-In

This section provides an overview of the extension point used by the TurboMove plug-in, the plug-in architecture and the initialisation process implemented to create different move proposals for the refactorings.

4.1.1 Extension Point

To allow a user to interact with the plug-in using Eclipse *CDT* an extension point is used. The used extension point for all refactorings is *org.eclipse.ui.actionSets*. By hooking in, the application is able to register the implemented refactorings into the existing refactoring menu of Eclipse *CDT*. We add four actions to this menu, each covering one of the refactorings below:

Move to Class...
Move member-function to another class described in Section 3.1 on page 13.

Move to File...
Move member-function to another file described in Section 3.2 on page 32.

Move to Free-Function...
Move member-function to free-function described in Section 3.3 on page 37.

Move-Up to Parent Namespace...
Move-Up to parent namespace described in Section 3.4 on page 50.

To distinguish between the different refactorings, each is using a different action delegate sharing a common superclass *AbstractTurboMoveRefactoringActionDelegate*. Next to them, other classes necessary for refactoring instantiation and user interface loading are required, shown in Figure 3 on page 65. The class design of the action and delegate classes implemented to start the refactoring from the extension point is illustrated in Figure 2 on the next page.

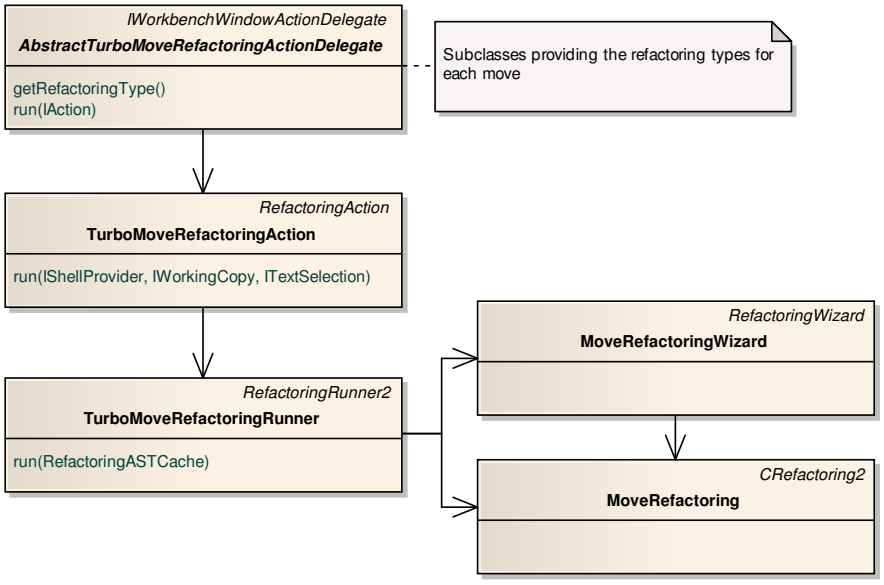


Figure (2) *Class Diagram of the Refactoring Extension Point Classes*

The interaction between these classes is fairly simple, shown in Figure 3 on the following page. The example uses the *move member-function to another class* refactoring action, however, the other transformations are using the same process to load the user interface and the refactoring itself when started from the Eclipse refactoring menu. For brevity, the abbreviation ”mmftac” is used to shorten the class name of the initial action. The creation process of the plug-in with the TurboMove related classes is described in-depth in Section 4.1.3 on page 67.

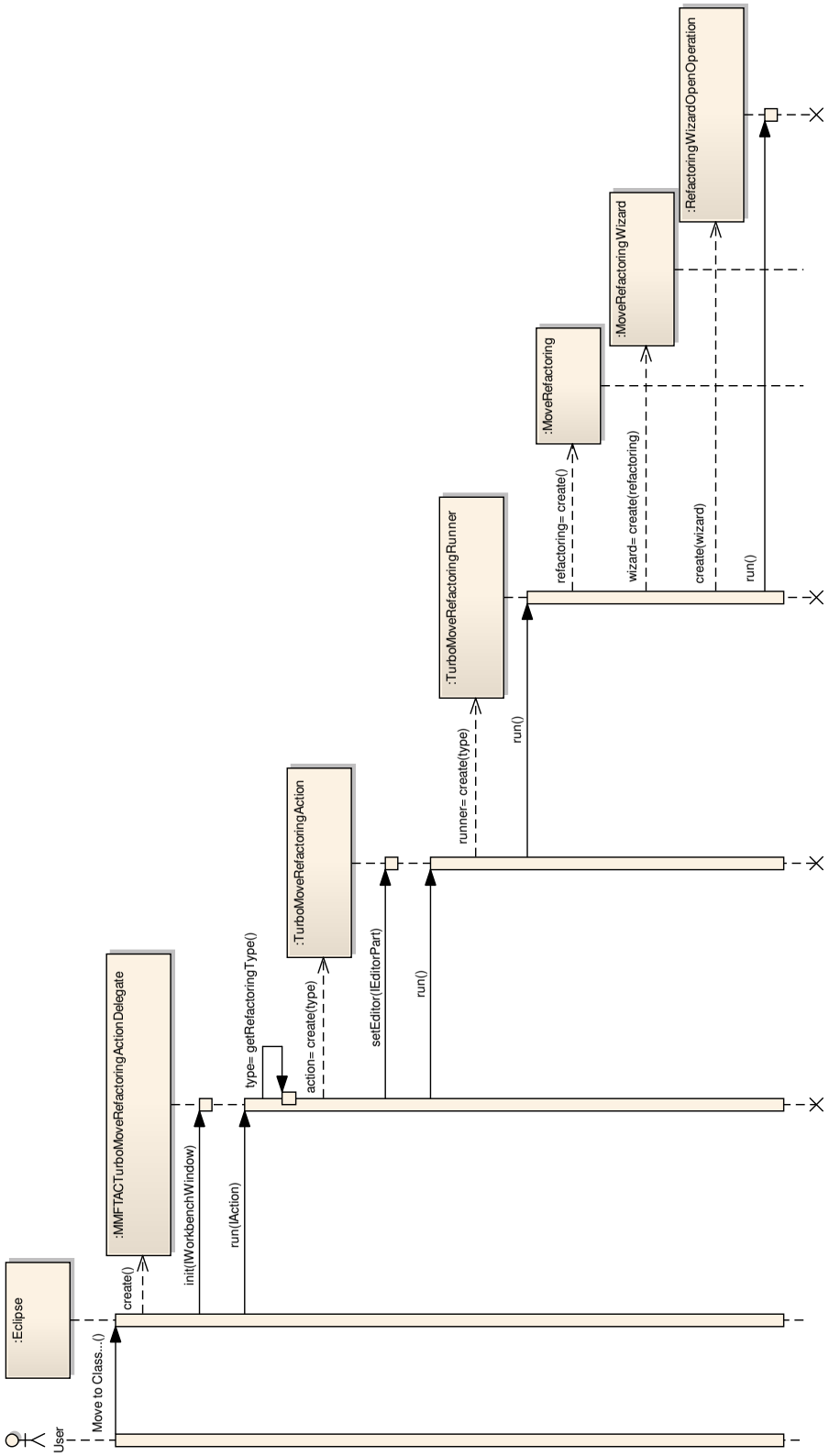


Figure (3) *Sequence Diagram of the Refactoring Extension Point Classes*

4.1.2 Architecture

The TurboMove plug-in contains several packages to group the functionality belonging together. The most important packages and their relationship are shown in Figure 4. For brevity, the package containing the user interface elements and utility packages are not shown.

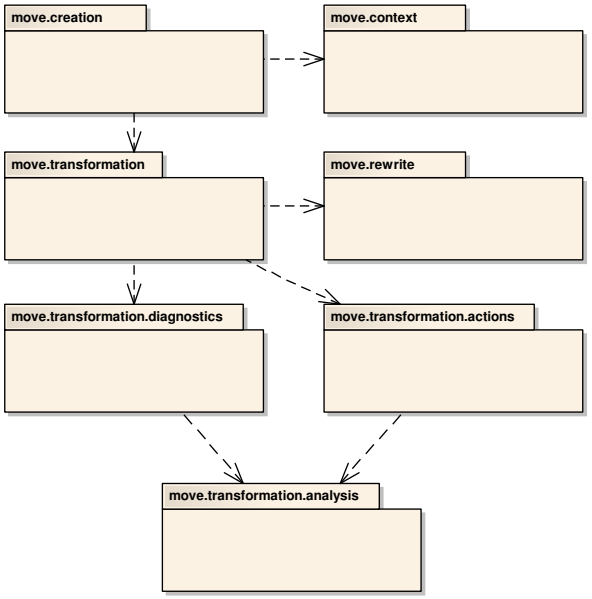


Figure (4) *TurboMove Packages*

The packages *creation* and *context* handle proposal generation and data calculation for the move refactorings. They are explained in depth in Section 4.1.3 on the next page. The rewrite package is an abstraction for the existing rewrite facility. They simplify the usage of the rewrites by caching existing rewrites and selecting the appropriate one for a change in an automated fashion. This package is explained in detail in Section 4.3 on page 78. Tasks related to collecting diagnostic messages and transformations based on the selected move proposal are located in the package *transformation* and subpackages. An in detail description of these functionality is given in Section 4.1.4 on page 72.

Since there are four major refactorings in this plug-in, we created a common class hierarchy to easily add or modify additional functionality. While *CRefactoring2* serves as a base class for all *AST* based refactorings, the implementing class of the TurboMove plug-in is used as a delegate to the real move implementation to be performed. The reason behind this design is the purpose of potentially having multiple move proposals available to a user. Thus, instead of implementing repeated extensions of the refactoring hook-in class, it is wise to split this functionality. The class diagram in Figure 5 on the following page illustrates this design.

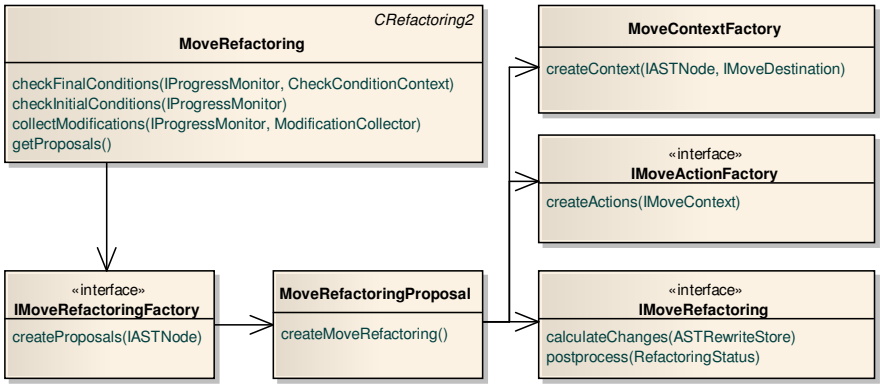


Figure (5) *Class Diagram of Delegation Mechanism to Move Proposals*

Implementations of the *IMoveRefactoringFactory* interface are used to provide proposals for a given refactoring type. For each of the existing transformations, a realisation of *IMoveRefactoring* and *IMoveActionFactory* exists along with a data context class created by the *MoveContextFactory* to provide shared informations for the transformation actions. This functionality is covered in detail in Section 4.1.3 and 4.1.4 on page 72. The interfaces shown in Figure 5 also picture the required extension points required to introduce additional refactorings to the plug-in (please note that these are not Eclipse extension points. They can be used only in the plug-in itself to add new refactorings).

4.1.3 Move Refactoring Initialization Process

After the selection of the move refactoring to perform and the process to create the move refactoring instance, the proposal creation and data calculation takes place. This Section will explain these two phases in detail.

Creation Process

The workflow of a refactoring is divided into three distinct phases, checking initial conditions, final conditions and collecting modifications. For the creation process, we will focus on the initial tests and the related tasks performed.

In Figure 2 on page 64 we have seen that each move refactoring has an assigned refactoring type. This type is an enumeration value available in *MoveRefactoringType*, passed through the startup process using the constructors of the action, runner and delegate classes explained in Section 4.1.1 on page 63 to the instance of *MoveRefactoring*. In the initial condition check phase, the move refactoring class calculates available proposals by selecting an implementation of the *IMoveRefactoringFactory* using the previously obtained refactoring type. For convenience, the lookup for this factory is performed using a factory provider class *MoveRefactoringFactoryFactory*. Each of the created proposals is initialized with a factory class to create the *IMoveContext* instance associated to the refactoring type along with an action factory creating the transformations to be performed. This design enables the plug-in to display proposals to a user, but delay the required data calculation until the proposal was

chosen. The process related to the initial condition check is shown in Figure 6 on the following page. For brevity, the abbreviation "mmftac" (*move member-function to another class*) is used to shorten the class names in the sequence diagram.

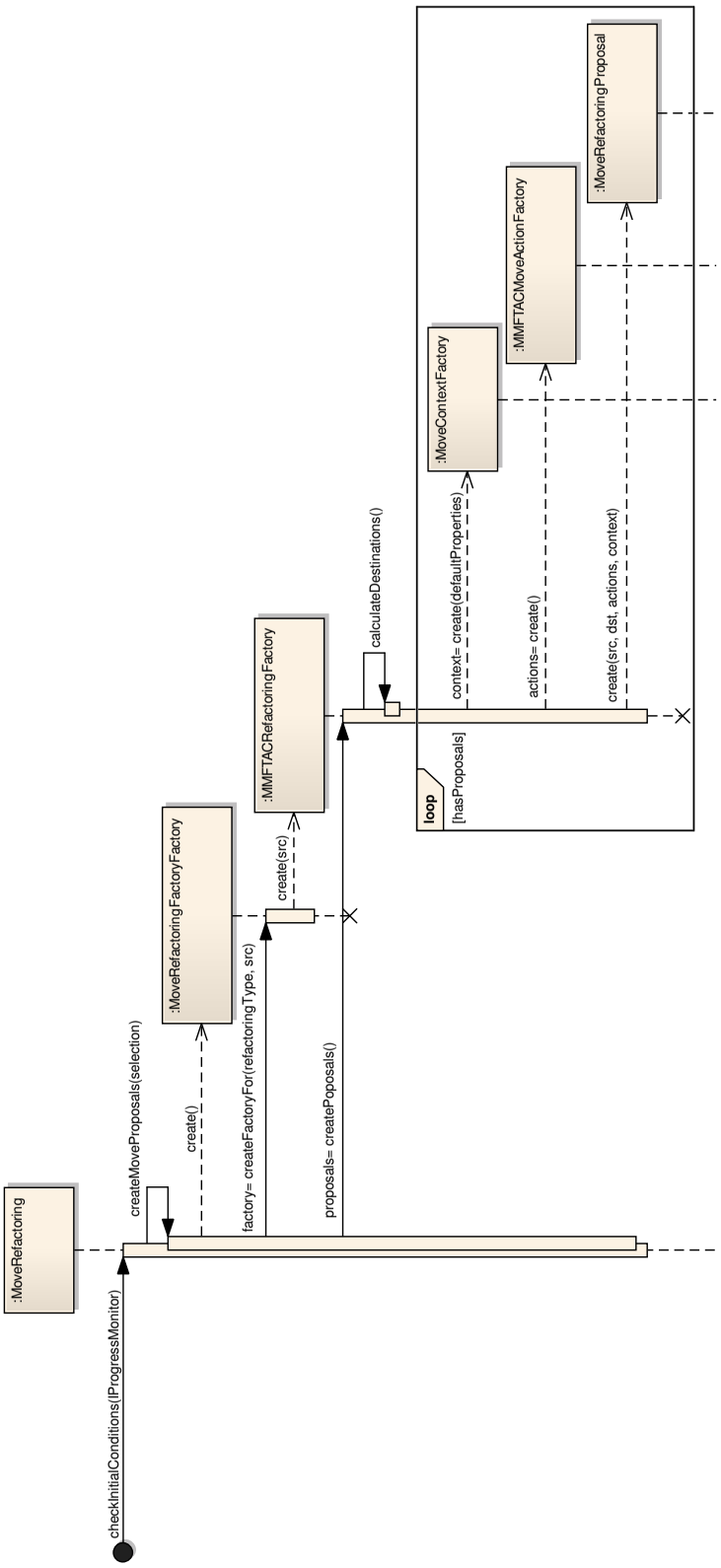


Figure (6) *Sequence Diagram of the Initial Conditions Test*

After the initial condition test phase, a user is allowed to select the desired move proposal and to set additional information. For example, in the *move member-function to another class* refactoring, there is an option to set the visibility adjustment style. These options can be set using the two enum classes *RefactoringPropertyType* and *RefactoringProperty*. The property type is used to indicate the feature to be set and the property itself serves as a choosable option. Applying the visibility example, it is possible to enable adding a friend declaration by setting the property *RefactoringPropertyType.VISIBILITY* to *RefactoringProperty.USE_FRIEND_VISIBILITY*. All properties are defaulted during the proposal creation process using a map of default properties in the implementing classes of the *IMoveRefactoringFactory* interface. This process is shown in Figure 7. For brevity, the refactoring property is displayed in shortened form.

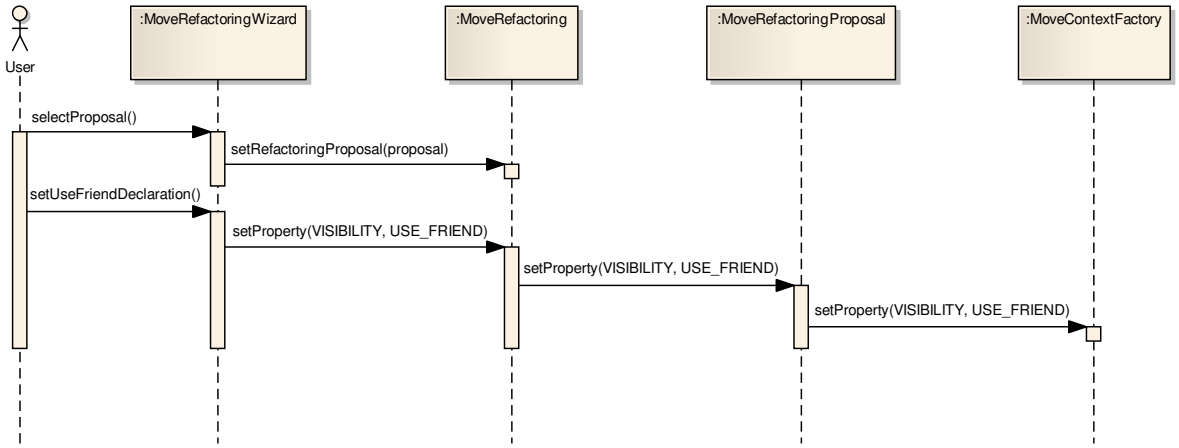


Figure (7) *Sequence Diagram of the User Selected Options*

In the final condition test phase, the selected proposal is used to create the real refactoring instance. This includes the data calculation described in the next Section and the move diagnostics, explained in Section 4.1.4 on page 72.

Calculation Process

The calculation process includes gathering data about the selected move proposal. This information is used by the transformations to perform the necessary changes to the *AST* and by the diagnostic classes to create feedback messages for a user about the refactoring. For this part of the process, we will focus on the final condition tests.

In the final condition check method, the instance of the refactoring class implementing the *IMoveRefactoring* interface is created. Each move refactoring has an associated implementation of this interface. The selected move refactoring proposal creates the move context and passes the instance of this context as well as the action factory to the move refactoring . This is illustrated in Figure 8 on the next page.

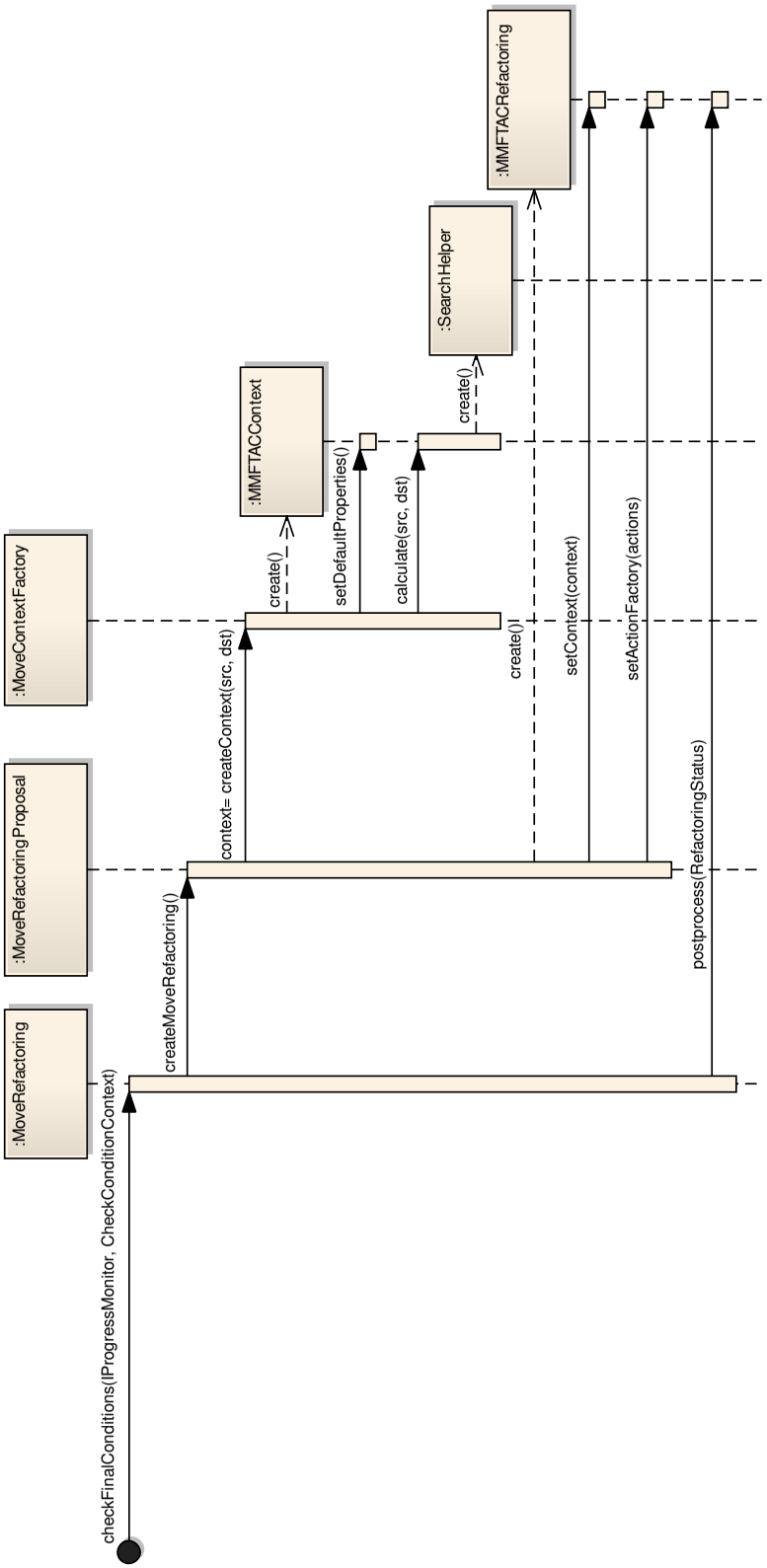


Figure (8) *Sequence Diagram of the Final Conditions Test*

The creation of the context includes the calculation of the required data for the move. For example, the context class of the *move member-function to another class* refactoring will lookup if a new parameter must be added to the moved member-function and if so, it calculates the optimal parameter passing mechanism (i.e. reference or const-reference). The context is also responsible for searching call-sites to change as well as other dependencies to solve. With this informations, the transformation and diagnostic classes are able to apply the changes.

4.1.4 Transformations and Diagnostics Architecture

In this Section, the design related to the transformations and diagnostics is explained in detail. We will do this in two parts. The first part will cover the class design of the transformation and diagnostics. The second part includes the occurrence of the diagnostics and transformations in the plug-in lifecycle.

Transformations and Diagnostics Architecture

All transformations share a common base interface *IMoveAction* with the methods *setContext(..)* and *apply(..)*. Each refactoring type has an associated factory implementing the *IMoveActionFactory* interface to create the correct instances to perform for the required changes. Basically, there are two types of actions, namely default actions and user selected actions. The default actions are applied independent of the configurations a user is choosing in the user interface. In contrast, the user selected actions depend on the refactoring properties set. For example, in the *move member-function to another class* refactoring, the option to use a friend declaration instead of visibility labels to change visiblities will force the action factory to yield a transformation to add this friend declaration instead of a label transformation.

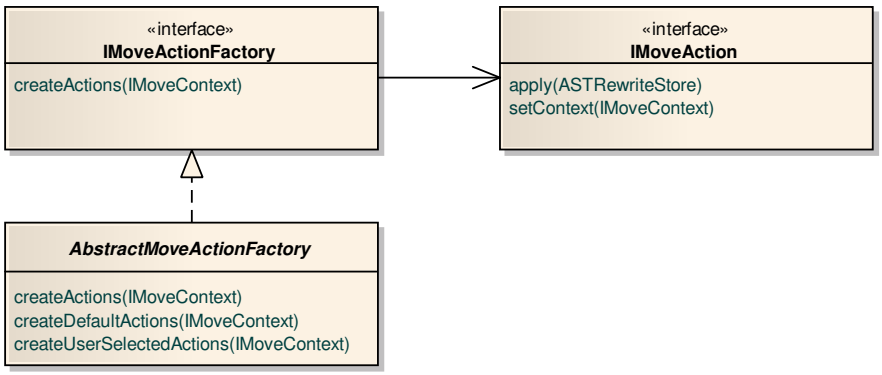


Figure (9) *Class Diagram of the Move Transformation and Factory Class*

The diagnostic classes use a similar approach. They implement the interface *IMoveDiagnostic* with the methods *setContext(..)* and *analyse(..)*. Likewise, they are separated into default diagnostics and user selected diagnostics. However, only one diagnostic factory exists for all move refactorings, because the default diagnostics can be created in the *IMoveRefactoring* instance and the user selected diagnostics are selected based on the user selected actions.

For example, in the *move member-function to another class* refactoring, the option to use a friend declaration instead of visibility labels to change visibilities does not require diagnostic messages for changed visibilities.

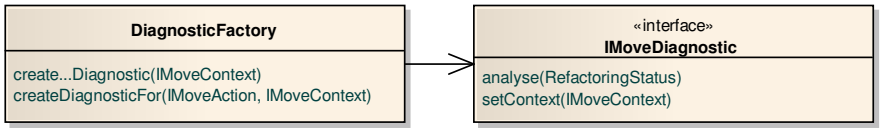


Figure (10) *Class Diagram of the Move Diagnostic and Factory Class*

The method *create...Diagnostic(..)* in this diagram serves as a placeholder for all other factory methods to create diagnostics.

Transformations and Diagnostics in the Plug-In Lifecycle

In the TurboMove plug-in lifecycle, the transformations and diagnostics are located in two different methods. The diagnostic messages are collected using the *postprocess(..)* method shown in Figure 8 on page 71. The method loads all default and user selected diagnostics and collects the status messages using the passed refactoring status. The plug-in is designed to allow transformations even with error messages in the final conditions. Although, we will not recommend to do this in most situations, the decision to continue is free to the user of the plug-in.

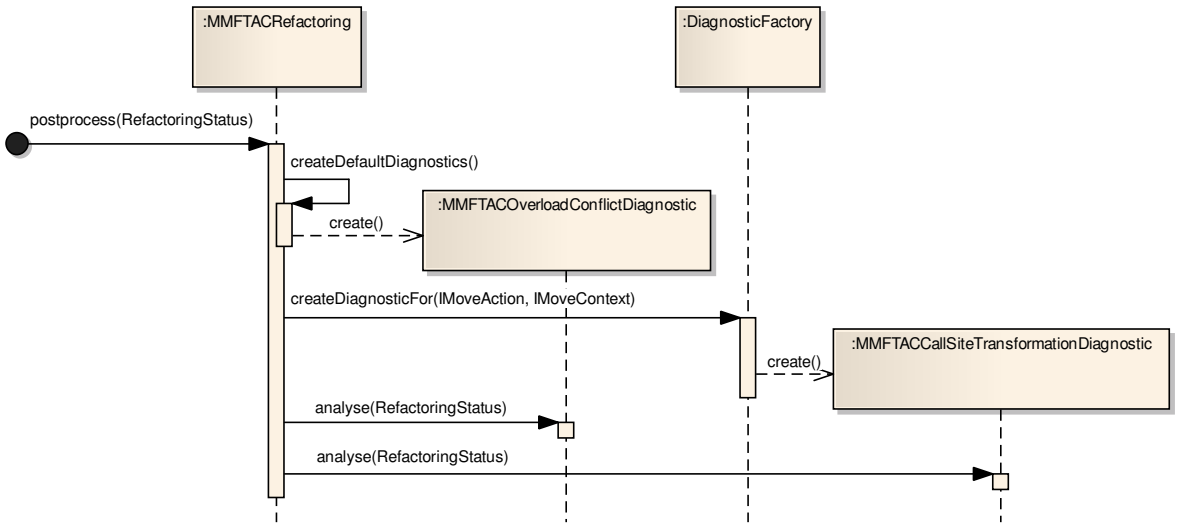


Figure (11) *Sequence Diagram of the Move Diagnostic and Factory Class*

Please note that Figure 11 only shows an example of the creation of diagnostic classes. The implementation does not directly instantiate the illustrated diagnostic implementations, but is using a loop to collect all default and user selected diagnostics. However, for understandability the example shows the instantiation explicitly.

The transformations of the *AST* are collected in the collect modification phase of the refactoring. The corresponding method in the TurboMove plug-in is *calculateChanges(..)* implemented in realisations of the *IMoveRefactoring* interface. The necessary transformation implementations are created using the *IMoveActionFactory* implementation of the chosen refactoring. The instance of this class is set during the move proposal generation process and provides methodes to create a list of actions to modify the *AST*.

4.2 Important Transformations

In this Section, the most important transformations for each of the implemented refactorings are explained. All the described changes are based on the described transformations in 3 on page 13, but from an implementation point of view. Please note that for some refactorings there is no description available, since the performed changes are easily understandable without explanation.

4.2.1 Move Member-Function to another Class Transformations

The *move member-function to another class* refactoring has two difficult transformations which will be explained in the following Section. The first describes changing the visibilities of existing dependencies to the originating type by adding visibility labels. The second transformation explained is the modification of the namespace the definition belongs to.

Visibility Transformation

In Section 3.1 on page 13, we explained the used mechanism to change visibilities of members of the originating type of the move used in the relocated member-function. From an implementation point of view, this is not a very easy task. If multiple member visibilities must be changed, it may be necessary to only open one public label for all these declarations, instead of adding a label for each declaration.

```
/**================================================
Example of visibility change for multiple declarations
================================================**/
// Before Transformation
struct A {
  // ...
private:
  void foo(); // change to public
  void bar(); // change to public
  void foobar();
};

// After Transformation
struct A {
  // ...
private:
public: // one label for two declarations
  void foo();
  void bar();
private:
  void foobar();
};
```

Listing (55) *Changing a Set of Declarations to Public*

In addition, there may are declarations with multiple declarators. In some cases, these declarators must be split into multiple declarations, but the order must be retained to avoid breaking initializations by constructors. Finally, there may exist members which should not be changed. The visibilities of these declarations must remain unchanged.

The first task in this transformation is the lookup of the related dependencies and a preprocessing step. These calculations are performed by the context implementation of the *move member-function to another class* refactoring. The preprocessing step examines all existing declarators in the originating type and calculates the required and the actual visibility of these declarators. The required visibility is either equal to the actual visibility, if the declarator is not a dependency, or the required visibility is public, if the declarator is a dependency.

$\mathsf{VISIBILITIES} := \{\mathsf{public}, \mathsf{private}, \mathsf{protected}\}$
$\mathsf{DECLARATORS} := \{\mathsf{d}_1, ..., \mathsf{d}_n\}$
$\mathsf{DECLARATORS\_PREPROCESSED} := \{(\mathsf{d}_1, \mathsf{v}_{11}, \mathsf{v}_{12}), ..., (\mathsf{d}_n, \mathsf{v}_{n1}, \mathsf{v}_{n2})\}$
$\mathsf{v}_{i1} \in \mathsf{VISIBILITIES}$ (actual visibility)
$\mathsf{v}_{i2} \in \mathsf{VISIBILITIES}$ (required visibility)
$\mathsf{i} \in \{1, ..., \mathsf{n}\}$
n = Number of declarators

Now, the actual transformation class takes over the control of the process. The declarators are grouped according to the actual visibility a label introduces to the group. However, the order of the declarators is not changed. For example, if the first three entries are public, number

four is private and number five is public, this will yield three groups (public, private, public). Please note that successive labels with the same visibility yield a group for each label.

$$\begin{aligned}
&\text{DECLARATOR_GROUPS} := \{b_1, ..., b_n\} \\
&b_i := (v_{x1}, \{d_x, ..., d_y\}), \forall d_i \in \{d_x, ..., d_y\} : v_{x1} = v_{i1} \\
&i \in \{1, ..., n\} \\
&n = \text{Number of declarator groups}
\end{aligned}$$

These groups are used to calculate a continuous visibility state for all declarators. The declarator group sets the initial actual visibility for all states of a group. For each declarator in the group, the actual and the required visibility is compared. If the actual and required visibility is not equal, the actual visibility is changed to the required visibility, meaning that at this position, a label will be inserted and all follow up declarators will have this visibility as well. It is essential to use the visibility groups in this steps. The declarators with the actual and required visiblility are not sufficient. Consider the following situation: There are two visibility groups and both are private, because for each group a separate label was used. If the last declarator of the first group is changed to public, the actual visibility is adjusted to public as well. However, if the first declarator of the second group must be changed to public as well, the actual visibility is still considered to be public, therefore, no label is added even it is required.

$$\begin{aligned}
&\text{VISIBILITY_STATES} := \{s_1, ..., s_n\} \\
&s_i := (d_i, v_{(i-1)2}, v_{i2}) \\
&i \in \{1, ..., n\} \\
&n = \text{Number of visibility states}
\end{aligned}$$

Note that $v_{(i-1)2}$ can be equal to v_{i1} if all previous visibilities were sufficient or if the actual visibility was restored to the group visibility.

The last step before the actual transformation can take place is an additional grouping of the declarator visibility states by their respective owner. By doing this, the transformation is aware of declarations with multiple declarators and can split them accordingly.

$$\begin{aligned}
&\text{DECLARATION_GROUPS} := \{e_1, ..., e_n\} \\
&e_i := (\text{owner}(d_x), \{s_x, ..., s_y\}), \forall s_i \in \{s_x, ..., s_y\} : \text{owner}(d_x) = \text{owner}(d_i) \\
&i \in \{1, ..., n\} \\
&n = \text{Number of declaration groups}
\end{aligned}$$

The transformation itself is fairly easy to perform. Each declaration is checked whether it has multiple declarators. If this is not the case, the underlying declarator visibility state is checked whether the actual visibility is equal to the required. If the visibilities do not match, a label is added. If multiple declarators exist, the declaration is split at the declarators with insufficient visibilities and labels are added as well.

Namespace Transformation

Changing the namespace membership of a member-function is not an easy task as well. As explained in Section 3.1 on page 13, the surrounding namespaces of the original definition are

closed before the definition and reopened after the definition. In addition, the definition name is changed to be fully-qualified. The complexity of this transformation arises from the tree structure used to represent the source-code. In the *AST* it is easy to relocate a single node or a sub-tree, however, closing and reopening surrounding namespace in an *AST* requires splitting this namespace-nodes, potentially over multiple levels. Hence, we have to implement a transformation which can apply these changes easily.

First of all, a simple check can be added to the transformation to potentially avoid changing namespaces at all. For this, the actual namespaces are compared with the new namespace qualifiers. If the namespace of the destination type is equal to the namespace of the originating type, no changes are required. If the destination type is in a different namespace than the originating type of the move, the transformation calculates a path to the definition to change. This path is a list of integers pointing to the next child in the tree until the definition is encountered. With this path, a recursive implemented function will split the tree at the index positions into two parts, namely the ”before” the definition and the ”after” the definition part. This is done by removing all nodes from the current namespace up to the current index, and reinsert them into the ”before” part. Than the process continues on the current index node, which is either another namespace or the final node. A check whether the final node is reached serves as the base case for the recursive call to finish. The only thing left to do after the recursive call terminates is to insert the recreated ”before” part before the original namespace and afterwards insert the function definition to move before the original namespace as well.

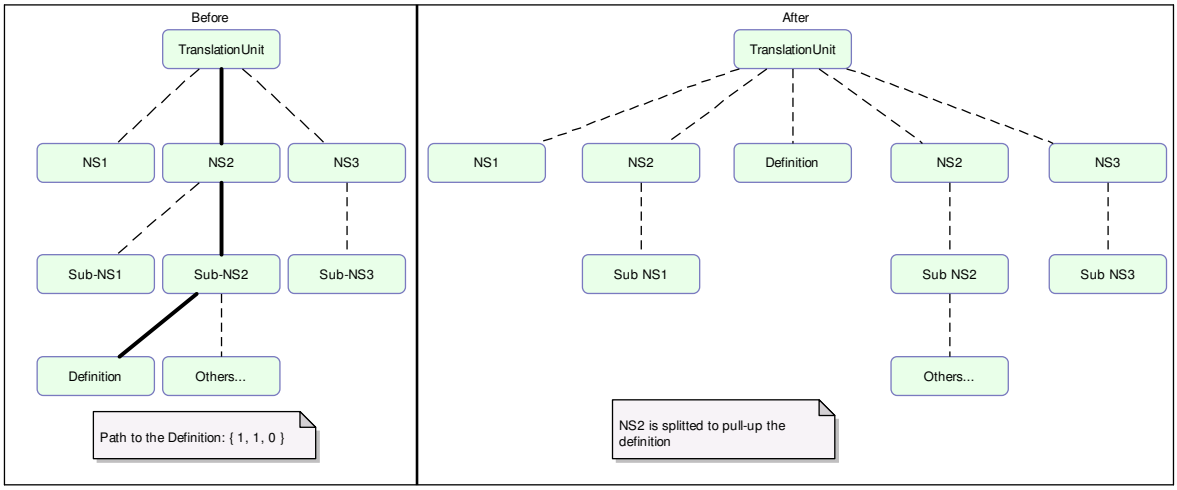


Figure (12) *Namespace Transformation Before and After*

4.2.2 Move Member-Function to Free-Function Transformations

The only difficult transformation for this refactoring is the visibility adjustment of the dependencies to members of the originating class. However, these changes are similar to those explained in the visibility transformation of the *move member-function to another class*

refactoring. Therefore, we refer to Section 4.2.1 on page 74.

4.2.3 Move-Up to Parent Namespace Transformations

The *move-up to parent namespace* refactoring has one important transformation which is explained in this Section. It describes the changes required to move the type or free-function to a new namespace. Although we already encountered this problem in the *move member-function to another class* transformation, we employ a slightly different approach here.

Namespace Transformation

The problem to change the namespace membership of an *AST* node is nothing new. We already have done this in the *move member-function to another class* refactoring described in Section 4.2.1 on page 74. However, in this transformation the node to change is moved-up only one level. Therefore, we can slightly adjust the original algorithm to successfully complete the task.

Since we know the node to move-up and the surrounding namespace, we can easily calculate the position of this node in this namespace by looking through the existing declarations until the node to move is encountered. Afterwards, a copy of the surrounding namespace without owned declarations is created and each declaration after the calculated index is removed from the original namespace and inserted into the copied namespace. Finally, the node to move is removed from the original position and inserted below the previously surrounding namespace. The newly created namespace is added below the old namespace as well, but this action must be performed before the node to move is inserted. Otherwise, the order is incorrect because the new namespace will be inserted before the node to move. The important thing for this transformation is to use the rewrite facility to add the removed declarations to the new namespace. Creating a copy of the nodes is possible as well, however, the copies cannot be changed by other transformations since they are new instances. Thus, adjusting existing using declarations below the node to move would not be possible. It is noteworthy to mention that the new namespace must be added to the *AST* first using the rewrite facility. Otherwise, the new namespace is not recognised as part of the *AST* and the transformation fails.

If the node to move is a type definition, all members of this type must be moved up as well. Basically, this works the same way as for free-functions or forward declarations of types. However, there are situations where member definitions are in sequence, similar to the declarations in the visibility label transformation explained in Section 4.2.1 on page 74. In this case, the members are moved-up as a group to avoid unnecessarily closing and reopening the namespace multiple times.

4.3 Extended Rewrite Facility

The refactoring infrastructure of Eclipse *CDT* allows rewriting the source-code using *ASTRewrite*s. Every time an insert or replace operation is performed on the *AST*, a new *ASTRewrite* instance is created. The root used for this rewrite is the changed node. All changes to nodes in the subtree of a previously changed node must be registered on the

correct rewrite. Otherwise the modification may not perform correctly. However, manually tracking the correct rewrite is hard and if an incorrect rewrite is used, finding the cause of the erroneous transformation can be difficult. In addition, the task of writing the transformations itself should be the challenge, not dealing with multiple rewrite instances. Therefore, we created an extension for the existing rewrite facility to simplify using the rewrite facility during development work.

4.3.1 Exploiting Rewrite Roots

The real problem behind using rewrites is the selection of the correct rewrite for a given transformation. Every time an operation is performed on the *AST* a new rewrite is created with a new root node. The root node used for the rewrite is easy to determine, because the changed node (i.e. the newly added node) always becomes the root. The only exception is the remove, however, removing a node from the *AST* does not create a new rewrite anyway, thus it can be ignored. With this information, we created a class to store rewrites and collect transformations. Once the transformations are applied, the correct rewrite is selected based on the root lookup node provided by the transformations. The class design is illustrated in Figure 13.

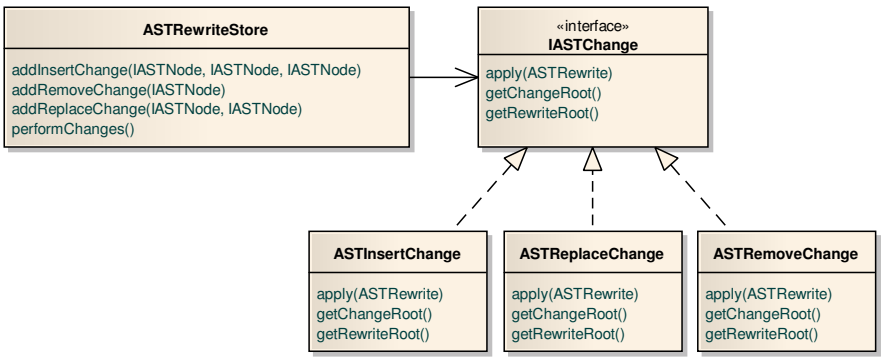


Figure (13) *Class Diagram of the Extend Rewrite Facility*

The changes are registered in the *ASTRewriteStore* by using the *add...Change(..)* methods. Afterwards, all changes can be performed using the *performChanges()* method. This method will use the *getRewriteRoot()* method to retrieve the lookup key for selecting the correct rewrite on which the change is applied. The new rewrites created by the changes are registered in the store by using the return value of the *getChangeRoot()* method as a lookup key. This process is shown in Figure 14 on the next page. Please note that this sequence diagram uses an imaginary "Transformation" class to illustrate different transformations registering actions.

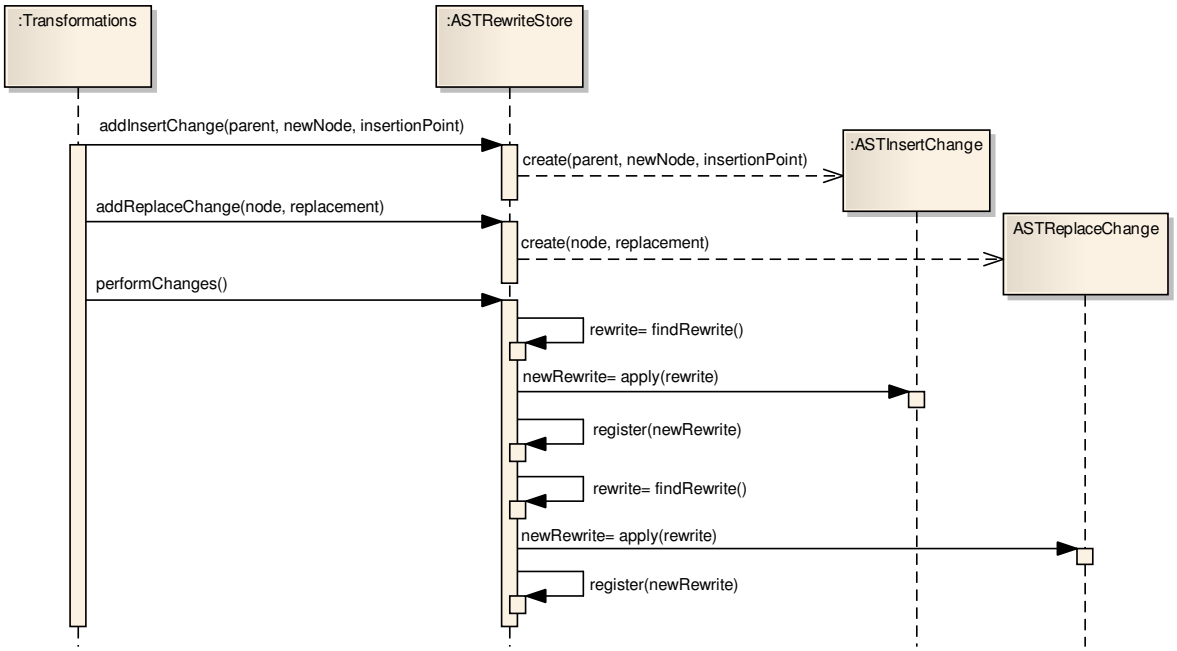


Figure (14) *Sequence Diagram of the Extended Rewrite Facility*

4.3.2 Open Issues

Although the extended rewrite facility simplifies the usage of the rewrite facility, not everything can be solved this way. For example, replacing a node in a subtree of a previously replaced node is not possible. However, this problem does not depend on the usage of the extended or the normal rewrite facility. Although, it could be possible to solve this problem by maintaining a node history based on the changes, but we did not implement this functionality. This problem is likely to exist due to a design problem not related to the rewrite facility, introduced by a erroneous order of changes added to the rewrite store. Therefore, we do not recommend an implementation of such a functionality either. Another problem of this implementation is the lack of functionality to register file changes (i.e. *CreateFileChange*), however, this can easily be added by creating an appropriate method in the rewrite store delegating to the modification collector.

4.4 User Interface

In the TurboMove plug-in there were several decisions available to implement a user interface. In the following, the possible interaction and the underlying decision for this implementation is explained.

4.4.1 Refactoring Menu

First of all, we had to ensure that a user is not overwhelmed with the available options of the refactorings. The initial decision which refactoring to perform must be easy. Based on this, we created a refactoring menu entry for each of the implemented refactoring. This way, a user has an overview of the possible transformation types available and can chose the appropriate move.

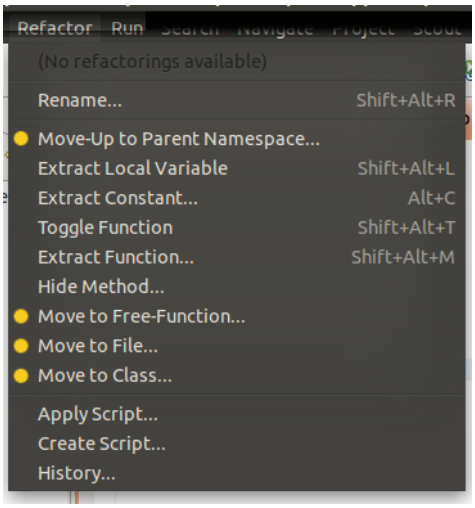


Figure (15) *Entries in the Refactoring Menu*

4.4.2 Refactoring Pages

The next step in the process is to allow a user to select the move proposal to perform. Since every refactoring in the TurboMove plug-in provides proposals, this task is equivalent for all transformations. Thus, the same user interface is used. However, some of the refactorings such as *move member-function to another class* require additional informations. For example, the ”free” move of the *move member-function to another class* refactoring requires specifying the desired move destination. Hence, we created an extended version of the proposal selection interface which allows to select a move destination based on the proposal selection. These two user interface types are shown in Figure 16 on the following page and 17 on the next page.

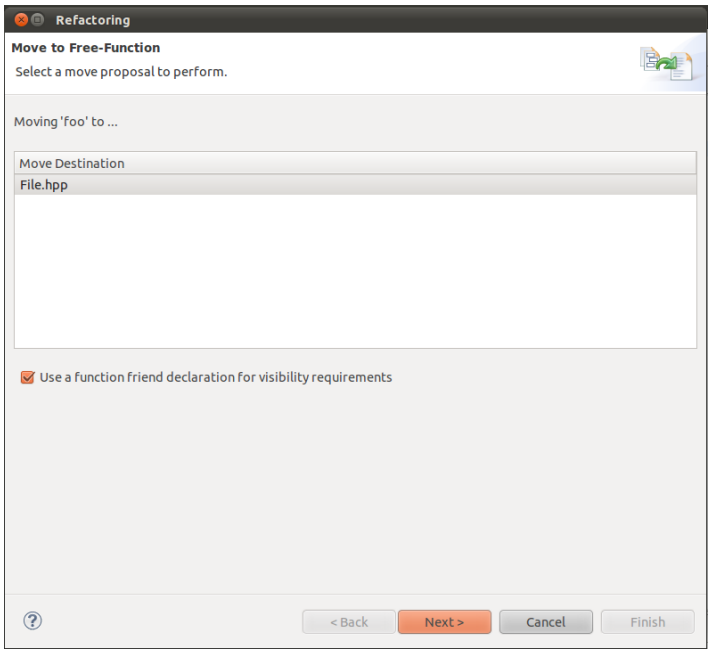


Figure (16) *User Interface with Proposals only*

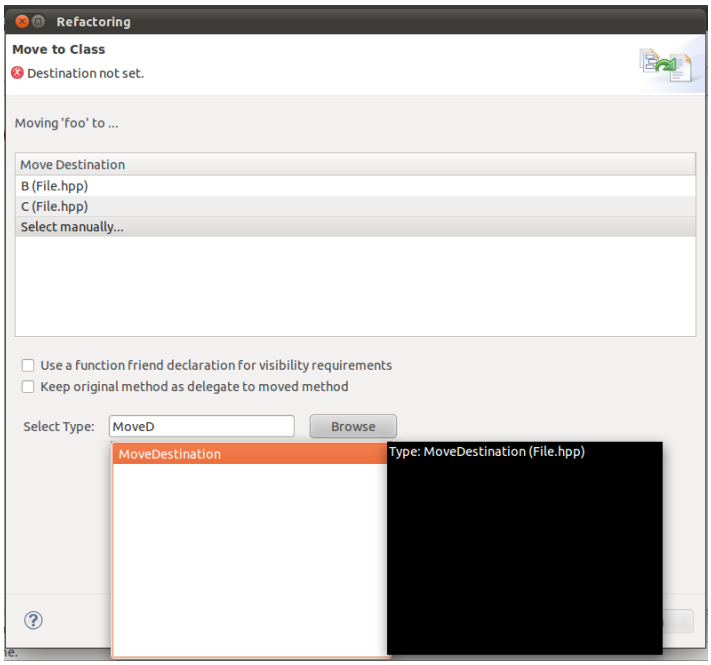


Figure (17) *User Interface with Proposals and Destination selection*

The following refactorings use the user interface which only allows the selection of a proposal

and options:

- Move member-function to free-Function
- Move-Up to parent namespace

The user interface including the destination selection dialog is used by the following refactorings:

- Move member-function to another class
- Move member-function to another file

Of course, the shown checkbox options can differ between the refactorings.

5 Conclusion

In this Section, we will present the project result along with the existing problems. In addition, an outlook for possible future work based on this master thesis is given. We conclude with a personal review of the project including personal impressions gained during the last weeks.

5.1 Project Results

Looking at the project goals in Section 1.3 on page 2 nearly goals of the project have been achieved. We have given an overview of possible move refactorings grouped into variable-, type- and function-based categories including an overview of potential problems that can occur. The TurboMove plug-in implements four of the evaluated moves which we have considered to be the most valuable. These refactorings are capable of giving a user proposals for a move destination together with different options to configure the transformation to perform the desired tasks.

The first implemented refactoring is capable of relocating a member-function declaration to a chosen destination type. By adjusting all existing dependencies on the in- and outside of the moved function, the compilability is retained. In addition, different strategies to adjust visibilities by using labels or a function friend declaration, or to keep the original function declaration and definition as a delegate are implemented.

As an addition to the *move member-function to another class* refactoring a refactoring to relocate a function definition into a selected destination file was created. Thereby, a user can select either existing files or create a new file as the desired move destination. To ensure compilability, existing includes in the originating file are copied to the target file as well.

Converting a member-function into a free-function is the task solved by the third refactoring. Similar to the *move member-function to another class* refactoring, all in- and outside dependencies are changed to retain compilability. Furthermore, selecting different visibility adjustment strategies is possible as well.

To change the namespace membership of a type or free-function a refactoring was implemented, which moves the chosen type or free-function one level up in the namespace hierarchy. The transformation will change all dependencies and call-sites by considering *Argument Dependent Lookup* (*ADL*) and namespace hierarchies making sure as less as possible changes are made to the source-code. To avoid call-site changes, a user can select an option to add a using declaration in the originating namespace pointing to the moved type or function.

Next to the implemented refactorings, we established higher level concepts for move refactorings. By separating the transformation into a logical and a physical move, it is possible to heavily reduce existing move options, but retaining flexibility. This makes the refactorings easier to use. The concept of retaining the position of the moved code if possible along with the minimal amount of changes paradigm allows a user to easier understand the result of

the refactoring. From our point of view, this is a fundamental part of a refactoring. Various changes to the source-code induced by changing positions and adding or removing elements decreases the understandability of the refactoring. Hence, users tend to not use the full potential of a refactoring, or they do not use it at all.

Implementing a facility to automatically recognise potentially moveable code could not be implemented due to time issues. However, the created refactorings are capable of generating proposals for a move. These proposals could easily be adapted to [Ecl12c] to provide move proposals on potentially ”bad” code.

5.2 Existing Problems

This Section explains open issues and potential bugs which could not be solved during the project.

5.2.1 Dependent Visibilities

In the *move member-function to another class* and *move member-function to free-function* refactoring described in Section 3.1 on page 13 and 3.3 on page 37, a mechanism to change visibilities is described. By adding visibility labels or a function friend declaration to the originating type it is possible to retain compilability if the moved function requires access to non-public elements. However, this may not be sufficient. This problem is illustrated in Listing 56 on the next page. Please note that the example does require another setup to run correctly, due to mutual dependencies. However, for brevity, this is left out.

```
1 /**================================================
2 Example of a dependent visibility introduced by
3 a friend declaration allowing to invoke 'bar'
4 ================================================**/
5 // A.h
6 struct A {
7   void foo() {
8     myB.bar();
9   }
10 private:
11   B myB;
12 };
13
14 // B.h
15 struct B {
16   // ...
17 private:
18   void bar() {
19     // ...
20   }
21   friend void A::foo();
22 };
```

Listing (56) *Dependent Visibilities Problem*

If we move the member-function *foo* to another class or to a free-function, an additional parameter of type *A* is added to the moved function to invoke the function *bar* on *myB*. For this, the visibility of *myB* is changed. However, this is not sufficient. Invoking *bar* in *foo* is only possible because this function is a friend function of the type *B*. Therefore, the adjusted invocation in a free-function or in another type will fail. We have not solved this problem in this thesis, but it could be done by adjusting the function friend declaration of the type *B*. If a complete friend declaration instead of a function friend is used, it may be necessary to add an additional friend declaration to satisfy the dependencies.

5.2.2 Template Call-Site Lookup

Moving a function template or type is desirable for all move refactorings as well. However, adjusting the call- or usage-sites of these templates is difficult. Eclipse *CDT* is not always able to resolve a template call-site to the appropriate template definition [Thr11]. Thus, applying the necessary transformations is not always possible. While this problem may be solved in future Eclipse *CDT* releases, it remains unsolved in this master thesis.

5.2.3 Undo File Creation

The *move member-function to another file* refactoring allows a user to create a new file as the desired move destination. However, there is an existing problem related to this task in the user interface of this move. If a user creates a new file using the plug-in user interface

and continues to the refactoring preview, but closes the page without applying changes, the new file is created, but not deleted. The reason for this is the used mechanism to create a new file in Eclipse *CDT*. The plug-in employs the *CreateFileChange* class to provide a new file, however, to allow an undo operation in the refactoring workflow, this change must be registered in the *ModificationCollector* instance of the refactoring. Unfortunately, the calculation process described in Section 4.1.3 on page 67 requires the translation unit and the underlying file to exist already, but the modification collector is not available in this phase of the refactoring. This problem could be solved by delaying the context calculation process. However, the diagnostics performed in the final condition test phase require the context data. Thus, this was not possible to solve.

5.2.4 Unary Expression Overload Method

In the *AST* of the Eclipse *CDT*, there are two main classes to deal with in the context of operators. *ICPPASTBinaryExpression* is used to represent binary operators and the interface *ICPPASTUnaryExpression* is used for unary operators. To find the declaration of a given operator the binding of this operator must be resolved and the position of the node is looked up using the index. Retrieving the binding of an binary operator is easy. For this purpose, the binary expression interface provides a method *getOverload()*. However, the unary expression interface does not have such a method, but the actual implementation of the interface *CPPASTUnaryExpression* has an existing public *getOverload()* method. In our opinion, the method was forgotten in the interface. For now, the plug-in uses a cast to the *CPPASTUnaryExpression* class instead of the interface to solve this problem, but in the future it may be possible to replace this with a cast to the interface. We already asked for a problem soluation for this issue in the Eclipse *CDT* mailing list.

5.2.5 Using Declaration Node Lookup

The *move-up to parent namespace* refactoring has the option to add a using declaration to the originating namespace. If a user chose this option, the call-sites remain unchanged, because the name still exists at the call-site. However, if this option is not selected, there may are existing using declarations that must be changed. Therefore, a lookup is performed whether there are existing using declarations to change. The search can be performed with the index by searching names for the binding of the type or function to move. If there are existing using declarations for the binding, an *ICPPUsingDeclaration* binding is returned. Unfortunately, resolving the node in the *AST* representing this using declaration is difficult. Normally, the index can be used again to find the declaration of a given binding. But this lookup will yield the function or type declaration to which the using declaration is pointing to instead of the using declaration itself. This behaviour is fairly useless for our purpose and we have not found a way to solve this problem using the typically used index functionality. Thus, we created a workaround to solve this problem. Each binding offers a method *getOwner()* returning the owner of the binding. For example, an *ICPPUsingDeclaration* binding can yield a function, a type, a translation unit or a namespace. With this owner binding, all declarations and definitions of the owner are searched and an *ASTVisitor* is applied to visit all names in the

AST. If the binding of a visited name is equal to the previously found *ICPPUsingDeclaration* binding, the node representing the using declaration was found. Unfortunately, this is not a good way to find the position of a node in the *AST*, but it seems to be the only way to deal with this problem at the moment.

5.3 Future Work

This Section outlines possible future works based on this master thesis, which could increase the value of the plug-in for a user.

5.3.1 Additional Refactorings

Although the plug-in contains four different refactorings, there are plenty of other transformations that could be implemented. In our opinion, a refactoring to move-down a type or free-function to a child namespace would be the best transformation to start at. Due to time constraints, we were not able to implement this refactoring, however, it could serve as an excellent complement to the *move-up to parent namespace* refactoring. In addition, the TurboMove plug-in only contains refactorings handling functions and types, but none of the variable based moves are implemented. We think implementing a set of these move types could make the plug-in even more valuable for an Eclipse *CDT* user.

5.3.2 Codan Support

The TurboMove plug-in provides move proposals for a given refactoring. However, the decision whether it is reasonable to apply a move refactoring to the code is the responsibility of a user of the plug-in. With Codan [Ecl12c] a proposal could be bound to a marker in the Eclipse editor if a given metric upper-bound is exceeded. For example, a Codan checker could evaluate a ratio between used parameter functionality and used ”this” dependencies of a member-function. If this ratio is ”bad”, the checker could propose the *move member-function to another class* refactoring.

5.3.3 Review the Refactoring Framework

The refactoring framework and especially the refactoring test facility can be hard to use. In this area, there are various options to develop improvements which would increase the usability and quality of all refactorings. This should also include a clear and understandable documentation of the provided functionality of Eclipse, such as *AST* explanations, best practices using the index and an infrastructure for tree-pattern recognition similar to the mechanism developed in [Kes10]. Such an infrastructure could be used to easily configure lookup and search strategies together with the results from an index query.

5.4 Personal Review

In summary, I am happy with the result of the project. I think the implemented refactorings are a great improvement for Eclipse *CDT* users and can help them in the day-to-day coding work. In my opinion, our goal to reduce the overall changes from the refactoring input to the

output is a great achievement as well. Not to mention the conceptual approach of separating a relocation into a logical and physical move. These paradigms probably will help developers creating refactorings in the future independent of whether they add more moves, or wish to tackle other refactorings.

While writing this master-thesis I was able to deepen my knowledge with Eclipse *CDT*. Especially the implementation of multiple refactorings and creating a design that can be used by all of the implemented transformations was a challenge for me. It was also a good experience from the theoretical work point of view. The problems to solve required carefully thinking about solutions that are easy, reasonable and appropriate to solve the task, but I was not bound to definite ”rules”. Instead it was still a very creative way of working for me.

In the beginning of the project, I spent more time for thinking and writing about possible moves and related problems compared to previous projects. I think this helped me a lot in getting a deeper insight and to foresee parts of the problems that can occur. The most important lesson learned for me is a better awareness of problem separation. If a problem is complex, you either do not understand it well enough, you need a more appropriate abstraction or another problem separation. Especially the separation of the logical and physical move made a previously very hard problem much easier.

I hope the TurboMove refactorings will find their way into the official Eclipse *CDT* release and I am confident that this will happen in the future.

5.5 Acknowledgements

I want to use the opportunity to thank various people that supported me during this master-thesis:

Peter Sommerlad, for the mentoring, not only for this thesis, but for all the projects during my master degree. He always provided me with valuable feedback and background in my work.

Thomas Corbat, for helping me out with Eclipse related problems that occurred during the project and of course for his great support in the review and correction of the final report.

My fellow students, for all the discussions we had, the sometimes necessary distraction and the laughs we had together.

A User Guide

In the following Sections, the requirements to use the TurboMove plug-in are described along with an installation and user guide.

A.1 Requirements

The TurboMove plug-in was developed for the Eclipse *CDT* Indigo release 8.0.0. While it may be possible to use the plug-in with older releases, it is not guaranteed to work with version before this release. You can download the supported release from the official project page [Ecl12a] or by using the update-site [Ecl12b] out of Eclipse.

Project Page: http://www.eclipse.org/cdt/downloads.php

Update Site: http://download.eclipse.org/tools/cdt/releases/indigo/

A.2 Installation

The plug-in can be installed using the provided update-site package. Until now, there is no update-site available in the web, however, this may change in the future. To install the plug-in from the update-site zip archive, use the Eclipse software install mechanism available in the "Help → Install New Software..." menu.

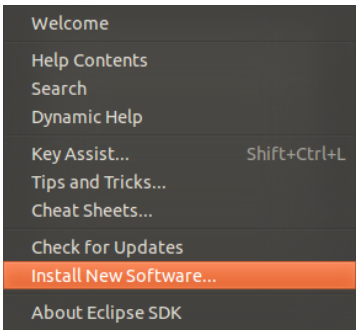


Figure (18) *Eclipse Menu to Install New Software*

This installation procedure is valid for a web update-site as well.

A.3 Refactoring Guide

Once the TurboMove plug-in is installed using the refactorings is fairly easy. The transformations are accessible in the refactoring menu of Eclipse shown in Figure 19 on the next page.

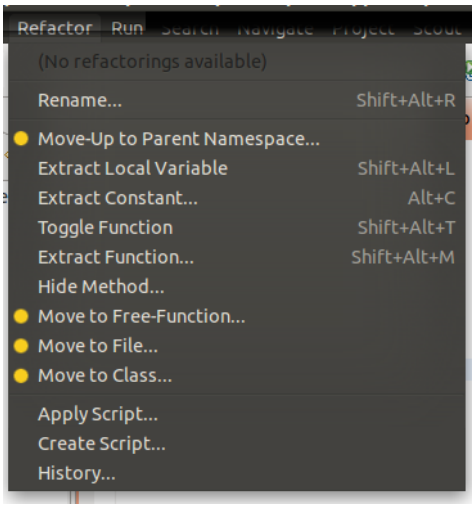


Figure (19) *Eclipse Refactoring Menu with TurboMove Refactorings*

To successfully start a refactoring a valid selection in the editor must exist.

- For the *move member-function to another class* refactoring, a member-function declaration must be selected
- For the *move member-function to another file* refactoring, a member-function definition must be selected
- For the *move member-function to free-function* refactoring, a member-function declaration must be selected
- For the *move-up to parent namespace* refactoring, a free-function declaration, type declaration or type definition must be selected

With a valid selection for a refactoring the transformation process can be started from the refactoring menu. There are two different refactoring user interfaces. The first allows you to select a move proposal and depending on the selected proposal to specify a move destination, shown in Figure 20 on the following page.

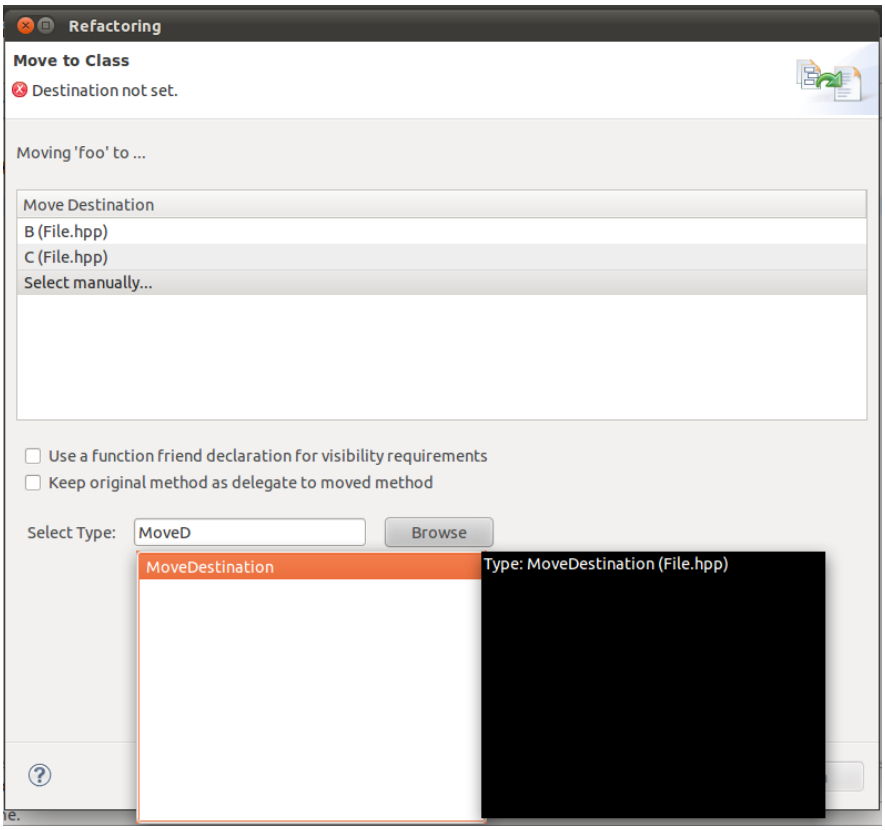


Figure (20) *Refactoring User Interface with Proposals and Destination Dialog*

The second wizard page only shows the available move proposals. An example of this interface is shown in Figure 21 on the next page.

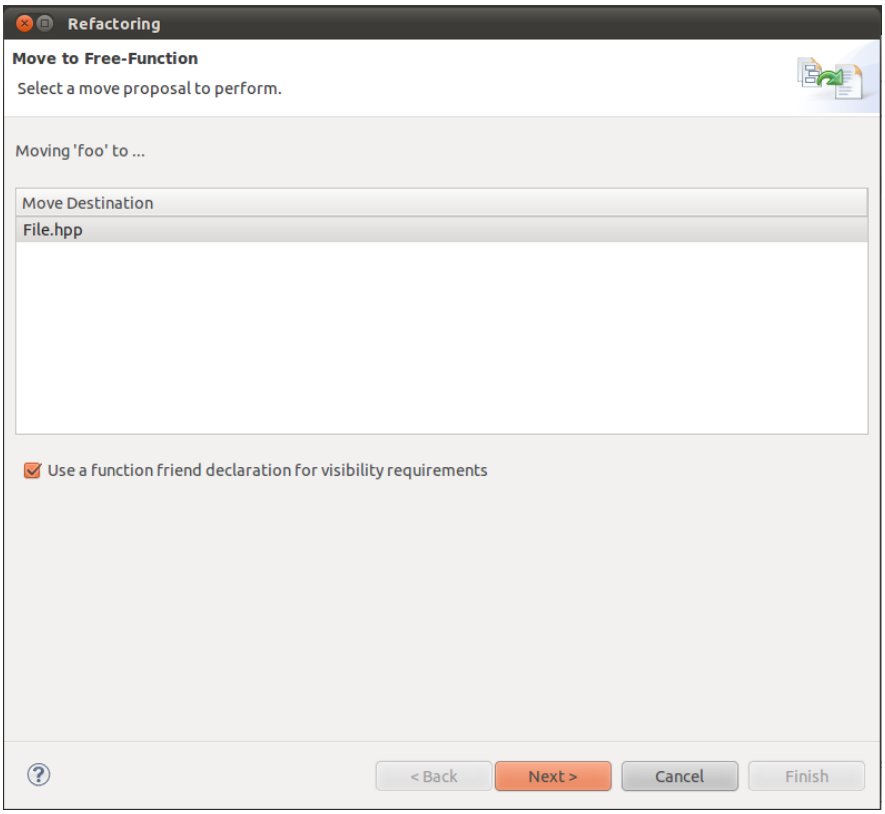


Figure (21) *Refactoring User Interface with Proposals*

Some of the refactorings allow to specify additional options for the transformation. They can be enabled or disabled using the checkboxes below the list of available move proposals.

- The *move member-function to another class* refactoring, allows to keep the original method as delegate to the moved method and to use a function friend declaration for visibility requirements instead of labels
- The *move member-function to free-function free-function* refactoring, allows to use a function friend declaration for visibility requirements
- The *move-up to parent namespace* refactoring, allows to add a using declaration in the originating namespace

After the options are set and the ”Next” button is pressed, the refactoring wizard will display the preview page comparing the original and the transformed code. Pressing the ”Finish” button will perform the actual transformation.

B Project Management

In the following Sections, we provide an overview of the project management of this master-thesis. This includes details about the project environment and the time spent for the project. In addition, an interpretation of the initial compared to the actual planning is made.

B.1 Project Environment

This Section describes the used system for continuous integration and the local development environment along with the used tools and software.

B.1.1 Continuous Integration Server

The continuous integration system used in this project was provided as a virtual server *sinv-56040.edu.hsr.ch* hosted at the University of Applied Sciences Rapperswil (*HSR*). The operating system of the virtual server is *Ubuntu 10.04 LTS 64-Bit*. To build the TurboMove plug-in and related artifacts the software listed in Table 1 is installed.

Software	Version
Apache	2.2.14
TeX Live	3.14
Hudson	1.376
Java Runtime Edition	1.6.0_20-b02
Maven	3.0
Trac	0.11.7

Table (1) *Installed Software on Build System*

B.1.2 Local Development Environment

The plug-in was created on a Lenovo T420s notebook with *Ubuntu 11.04 64-Bit* installed. For development, the software listed in Table 2 was used.

Software	Version
Eclipse Indigo	3.7.0
TeX Live	3.14
OpenJDK	1.6.0_22
Maven	3.0
gedit	2.30.4

Table (2) *Installed Software on Development System*

B.2 Project Plan

This Section covers the time used for this master-thesis. The project is rewarded with 27 *European Credit Transfer System* (*ECTS*) points. For each point, 30 hours of work are estimated. This gives a total of *27 * 30 = 810* hours for the project. In the beginning of the project, we planned with 22 weeks excluding two weeks for christmas and new years break. This results in 20 weeks working time for the project and an estimated workload of *810 / 20 = 40.5* hours/week.

B.2.1 Actual vs. Target Hours/Week

In Figure 22 the actual vs. the target hours/week are compared. It is noteworthy to mention that there are 21 weeks included in this chart, since the two weeks originally planned for christmas and new years break were partially used for working at the project.

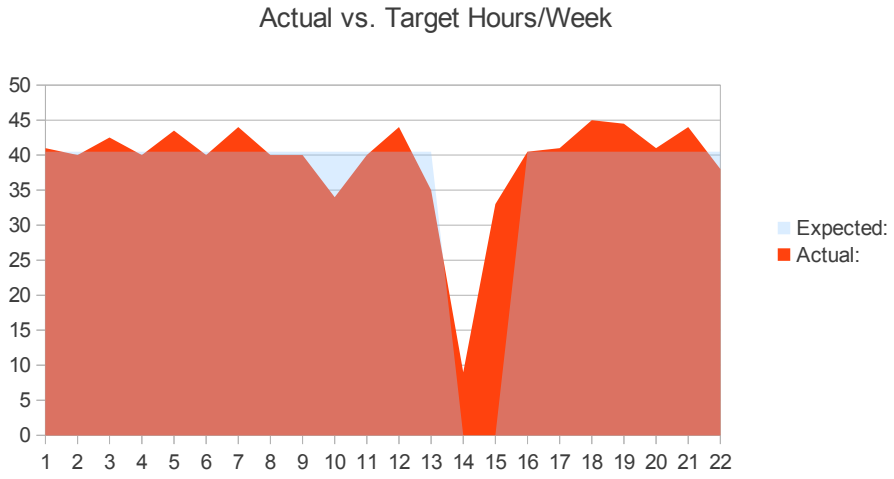


Figure (22) *Actual vs. Target Hours/Week*

B.2.2 Interpretation

The expected and actual weekly workload was roughly the same throughout the project, including some high and low peaks. The gap in week 14 and 15 is due to the initial plan of not working during christmas and new years break, however, some time was spent for the project during this phase. Overall, the time used for the project was 870 hours. This is about 7.5 % more time than actually planned.

B.2.3 Conclusion

Although the time spent for the project was higher than estimated, it was not exceeding the limit of acceptable workload for me. I think the additional time invested allowed me to

complete the project as good as possible and due to the opportunity to work over christmas and new years break, to deliver in time.

C Project Setup

This Section covers the project setup used in the TurboMove plug-in including the necessary dependencies and configurations using maven [Pro12]. This project setup is based on the project setup process described in [Thr10] and [Thr11]. For maven related configurations, we recommended to use the maven Eclipse plug-in [Son10].

C.1 Structure

The TurboMove plug-in contains six maven projects:

ch.hsr.cdt.turbomove
The maven parent project. All other projects are subfolders of this project.

ch.hsr.cdt.turbomove.app
A plug-in project containing the refactoring implementations and the user interface.

ch.hsr.cdt.turbomove.app.feature
Feature project to collect all plug-ins required to deploy the TurboMove refactorings using an update-site.

ch.hsr.cdt.turbomove.test
This plug-in project contains all test files along with the testing environment.

ch.hsr.cdt.turbomove.test.feature
Feature project for the test plug-in to run the tests.

ch.hsr.cdt.turbomove.updatesite
Update-Site project using the feature project of the TurboMove application plug-in providing an installable version of the refactorings

C.2 Creating POM Files

After the projects have been created, the maven pom.xml files have to be created. The easiest way to do this is to use Tycho [Son12], a maven plug-in using a manifest-first approach for creating and building an Eclipse plug-in. For this, open a terminal window, navigate to your parent project folder and execute the command shown in Listing 57. Please note that Tycho requires maven 3.0 or higher.

```
1 mvn org.sonatype.tycho:maven-tycho-plugin:generate-poms
      -DgroupId=<group-id> -Dtycho.targetPlatform=</cdt>
```

Listing (57) *Creating POM Files with Tycho*

The group-id is used as an identification name for your plug-in. Normally, the name of the parent project is used for this. The target-platform parameter is used to point to the Eclipse *CDT* installation for which the plug-in is developed. Make sure to use a location which is available on both, your development machine and the build-server. In this project, the following parameters are used:

Group-ID:
ch.hsr.cdt.turbomove

Target-Platform:
/usr/local/cdtmaster-8.0.0/

C.3 Maven Repositories

The next step is to specify the maven repositories required to build the Eclipse plug-in. For this, two entries have to be added to the pom.xml file of the parent project:

```
<repositories>
  <repository>
    <id>eclipse-indigo</id>
    <url>http://download.eclipse.org/releases/indigo/</
       url>
    <layout>p2</layout>
  </repository>
  <repository>
    <id>eclipse-indigo-cdt</id>
    <url>http://download.eclipse.org/tools/cdt/releases
       /indigo/</url>
    <layout>p2</layout>
  </repository>
</repositories>
```

Listing (58) *Eclipse Maven Repositories*

These repositories are used to load the required dependencies for the plug-in development.

C.4 Add Target Definition

The Tycho maven plug-in uses target-platform definition file to build the plug-in for a specific platform. This target specification should be added to the main application plug-in project. The name of this file must be equal to the containing plug-in name, but with the extension ".target". To create the target definition, navigate to "New → Target Definition" and enter the name of the file. In the TurboMove plug-in, the file name is *ch.hsr.cdt.turbomove.app.target*. In the target definition, both previously added repositories and the local installation of the target Eclipse *CDT* platform should be specified:

Local Installation:
/usr/local/cdtmaster-8.0.0/

Maven Eclipse Repository:
http://download.eclipse.org/releases/indigo/

Maven Eclipse CDT Repository:
http://download.eclipse.org/tools/cdt/releases/indigo/

The target definition can now be set to active using the ”Set as Target Platform” link on the top right corner in the editor.

C.5 Maven Dependencies

The plug-in requires several dependencies that have to be added to successfully compile. In the application project, the following dependencies exist:

- org.eclipse.ui
- org.eclipse.ui.ide
- org.eclipse.cdt.ui
- org.eclipse.cdt.core
- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.ltk.core.refactoring
- org.eclipse.ltk.ui.refactoring
- org.eclipse.jface.text
- org.eclipse.ui.editors

In the test plug-in project there are different dependencies to use:

- org.junit4
- org.eclipse.ui
- org.eclipse.cdt.ui
- org.eclipse.cdt.ui.tests
- org.eclipse.cdt.core
- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.ltk.core.refactoring
- ch.hsr.cdt.turbomove.app

These dependencies should be added using the ”Dependencies” tab in the manifest file of the plug-in projects. There may are more dependencies required depending on the plug-in to develop, however, the TurboMove plug-in only uses those listed above.

C.6 Exporting Packages for External Use

The separation between an application and a test plug-in is recommended. This way, it is possible to easily deploy the application without having unnecessary dependencies to the testing environment and the test classes are omitted as well. However, the test plug-in requires access to the packages of the application plug-in to run refactoring tests. Therefore, the required packages of the application plug-in project must be exported. This functionality is available in the ”Runtime” tab of the manifest file in the application plug-in.

C.7 Important Maven Commands

By using maven to build the plug-in project, different commands are available for testing, building the update-site and cleaning up the artifacts.

mvn integration-tests:
Run the maven tests. This command is normally used in the test plug-in project.

mvn clean:
Removes all previously build results from the specified output folders.

mvn package:
This command build the update-site for the plug-in project.

D Testing Environment

The TurboMove plug-in uses the extend testing environment originally developed in [Thr10]. The base infrastructure was developed in [Ema06] with additional explanations in [Ind10].

D.1 RefactoringTester

The testing environment uses the class *RefactoringTester* to create test instances using configuration files. Unfortunately, the refactoring tester employs the plug-in activator to load the base path for the lookup of test configuration files. In addition, there are Java restrictions for loading files from another plug-in. Therefore, the original refactoring tester must be copied to the test plug-in. Afterwards, the class must be changed to use the correct plug-in activator. This can be done by changing the *createReader(..)* method.

```
1 Bundle bundle = Activator.getBundle();
```

Listing (59) *Changing Activator in Test Plug-In*

To verify the correct activator is used, the import directives can be checked to point to the correct package in the test plug-in.

D.2 Test Configuration

The test configuration is a text file, normally located in a resource folder in the plug-in project. Traditionally, the file extension .rts (refactoring test) is used, however, this is not mandatory. The format of the test configurations is described in [Thr10], but we will point out some special fields added to the refactoring tests for the TurboMove plug-in.

userSelection:
This parameter is used to tell the refactoring which move proposal was selected.

destinationName:
The filename of the move destination.

destinationOffset/destinationLength:
Position of the destination node in the destination file.

expectedInitErrorCount:
Expected amount of errors in the initial condition test phase.

expectedFinalErrorCount:
Expected amount of errors in the final condition test phase.

expectedInitWarningCount:
Expected amount of warnings in the initial condition test phase.

expectedFinalWarningCount:
Expected amount of warnings in the final condition test phase.

D.3 Test Class

A refactoring performs three steps in the transformation lifecyle. First, the initial conditions are checked followed by user input. The second step is to check the final conditions. In the end, the change is generated and applied to the source-code. This lifecycle was already implemented for a testing environment in [Thr10], thus, we reuse it here. However, the class was changed to be able to work with the TurboMove refactorings.

D.4 Improved Tests

The TurboMove plug-in uses the improved tests described in [Thr10] including the XML based test description. We refer to the explanations made in [Thr10] for this Section.

D.5 Common Errors

Next to the common errors described in [Thr10], an additional problem that can occur was found. Normally, a test failure is displayed in the JUnit user interface with a blue failure sign. By clicking on the failure, the expected and the actual output can be examined. However, in some cases this is not possible, because the test failure is not displayed as a failure, but does not have a green success sign either. If this is the case, the configuration file related to this test specifies source-files having the same names as source-files listed in another test configuration. Changing the names of this source-files will solve this problem.

E Replace Complete Parameter with Parameter Data

In this Section, we will give a brief description of a refactoring that may be useful in collaboration with the *move member-function to another class* described in Section 3.1 on page 13 and *move member-function to free-function refactoring* described in Section 3.3 on page 37.

E.1 Motivation

The above mentioned refactorings add an additional parameter to the moved member-function if the function requires access to a member-variable or another member-function of the originating type. The new parameter is used as the new invocation owner of the existing dependencies. However, adding the complete type as a new parameter may not be reasonable. Consider the transformed example in Listing 60.

```
1 /**================================================
2 Example of unnecessarily passing 'Payment' completely
3 instead of 'amount'
4 ================================================**/
5 // DebitCard.h
6 #include "Payment.h"
7
8 struct DebitCard {
9   void settle(class Payment & newPayment) {
10    if(getBalance()<newPayment.amount) {
11      // refuse payment...
12    }
13    reduce(newPayment.amount);
14  }
15  double getBalance() {
16    return balance;
17  }
18  void reduce(double amount) {
19    balance-=amount;
20  }
21  double balance;
22  // ...
23 };
24
25 // Payment.h
26 #include "DebitCard.h"
27
28 struct Payment {
29   double amount;
30 };
```

Listing (60) *DebitCard Example after Move*

The parameter *Payment* is passed to the settle function, however, only the member-variable *amount* is used. In addition, amount only affects the member-variable *balance* in *DebitCard*, but is not changed in *Payment*. Therefore, it is reasonable to only pass *amount* instead of the complete payment.

E.2 Mechanics

Applying this refactoring involves several steps. Each step retains compilability, however, not completing a step can yield erroneous code.

1. Add a new local variable to the function initialized with the parameter access to replace
2. Replace the parameter accesses with the local variable
3. Apply the ”Add Parameter” refactoring [Fow04] for the local variable replacement
4. Assign the new parameter to the local variable
5. Apply the ”Inline Temp” refactoring [Fow04]
6. Apply the ”Remove Parameter” refactoring [Fow04] for the old parameter

E.3 Benefits

There are some benefits of applying this refactoring:

Lower Coupling:
The coupling of the code is reduced, because the function no longer has a dependency to the type previously passed as a parameter.

Understandability:
Passing only the data required for a function to perform a task is easier to understand, rather than passing a complete object.

Controlled Side-Effects:
By avoiding to pass a complete object, it is assured that depending on the parameter passing mechanism only the passed data will have side-effects.

E.4 Consequences

However, there are also consequences:

Lost Side-Effects:
By avoiding to pass a complete object, side-effects for this object may be lost (may depent on the parameter passing mechanism).

Breaking Dependent Code:
Third party users of your code can not update to the actual version of the source-code without breaking compilability.

F Refactoring Development in a Nutshell

Creating a refactoring plug-in is not always easy. It is not only the task to solve that can raise problems, sometimes the provided infrastructure is complicated to use or does not behave as expected. In this Section, we try to explain the most important parts of refactoring development. We hope that these "best practices" can help refactoring plug-in developers in the future.

F.1 Refactoring Hook-In

In this Section, we aim to explain the hook-in classes for refactorings and the associated lifecycle.

F.1.1 CRefactoring or CRefactoring2

To create a refactoring plug-in, a developer can chose between two classes, either *CRefactoring* or *CRefactoring2*. *CRefactoring* is older than *CRefactoring2* and should not be used anymore. With *CRefactoring2*, there is also a built-in caching infrastructure for *AST*s available per default.

F.1.2 Plug-In Lifecycle

A refactoring plug-in lifecycle consists of different phases. After the creation and initialization of the plug-in class, the initial conditions are tested. The initial conditions are used to determine whether a refactoring can be performed at all. This test is performed before the refactoring user interface is visible. For example, checking if the selection in the editor is valid for a given refactoring is normally a task performed in the initial condition test phase. The next part in the lifecycle is testing the final conditions. Final conditions are used to ensure the refactoring is possible with the configurations a user made in the refactoring wizard. All required informations should be calculated in this phase. Otherwise, it may be possible that the actual transformation of the source-code yields an error. This is not a good idea, because in the transformation process, it is not possible to give a structured feedback to a user. Only exceptions can be shown to a user without a chance of recovering from the error. The last phase in the refactoring lifecycle is the generation of the change and applying the change. The generated change is not yet made active in the source-code, but is shown to a user in the preview window of the refactoring. Only applying the change will modify the source-code. In this modification collection process, all changes should be applied. These changes should not yield any error if possible, however, this may not be possible due to using the index in this phase.

When developing a refactoring plug-in it is a good idea to always be aware of this lifecycle. Otherwise the risk of placing functionality into the wrong phase of the refactoring lifecycle is high.

F.2 The Index

In an Eclipse refactoring plug-in it is wise to use the index to perform lookups for nodes used in the transformation. However, it is necessary to think about the type of change to perform. We call this ”local changes” vs. ”distributed changes”. Local changes are refactorings transforming nodes related to the selection and the neighbourhood of this node. If the transformation only changes these nodes and no further information is required from other parts of the code, the index can be ignored. In contrast, distributed changes must use the index. For example, changing call-sites of a function requires looking up these call-sites using the index.

F.2.1 Index Lookup

The index can be used to perform different lookups.

findNames(IBinding, int):
Find names will search all names which resolve to the given binding. The search can be restricted to all, declarations, definitions, references and combinations of these flags. These flags are accessible using the constants defined in *IIndex*.

findDefinitions(IBinding):
This is the same as *findNames(..)* using the definition flag.

findDeclarations(IBinding):
This is the same as *findNames(..)* using the declaration flag.

findReferences(IBinding):
This is the same as *findNames(..)* using the reference flag.

findBindings(char[], IndexFilter, IProgressMonitor):
With this method, all bindings with the specified name are returned, independent of whether the bindings are equal. This is valuable if you try to analyse overload conflicts or similar tasks.

adaptBinding(IBinding):
This method will adapt a binding to an index binding. The reason for this is explained in Section F.2.2.

Please note that these lookup methods can return more candidates than you need. Normally, the returned results require further processing to select the appropriate results for the transformation. Considering the various ways C++ is offering to accomplish tasks, this can be inconvenient to do. In our opinion the best solution for this is to use a similar approach to the tree pattern matching described in [Kes10]. However, an implementation of this functionality available in a more general aspect for refactorings does not exist.

F.2.2 Comparing Bindings

Bindings are an important part of the *AST*. They can occur in two variations, either a normal binding or an index binding. A binding is used to ”bind” names together. For example, a

function name binding can be used to find call-sites of this function, because the bindings belong together. Bindings can be compared whether they are equal. This can be necessary to evaluate whether a given node must be changed or not. The important thing when comparing bindings is to always compare index bindings. If it is not sure whether a binding is an index binding, the *adaptBinding(..)* method of the index can be used for conversion. Comparing a normal binding with an index binding will yield false, even if they are equal.

F.2.3 Resolving Nodes

Once you found an entry using the index, it is necessary to find the related *AST* node. Normally, the results of an index lookup are bindings or index names. With an index name, you can load the related translation unit using *findElement(..)* on the project associated to the refactoring. With the *AST* of the translation unit, a node selector can be created using *getNodeSelector(null)*. Using the index name length and offset, the node selector is able to perform a lookup for the node.

F.3 Abstract Syntax Tree

The *Abstract Syntax Tree* (*AST*) is a representation of the source-code that can be used by the refactoring to perform transformations. The top node of an *AST* is always a translation unit containing several subnodes and subtrees, depending on the code in the originating source-file. In the beginning it may be hard to understand the structure of this syntax tree, however, Eclipes provides a view to show the *AST* for a given translation unit, accessible using "Window → Show View → DOM AST". This view is a good start to get a basic understanding of the structure and nodes that can exist in a translation unit.

F.4 Caching Syntax Trees

The process of creating an *AST* is separated into different phases. A source-file is analysed and an *ITranslationUnit* is created. This translation unit can be used to build an *AST* using *getAST(..)*. However, every invocation of this method will yield a new instance of the same syntax tree. This is not a desirable behaviour, because applying transformations to different *AST* instances of the same translation unit are likely to conflict. Fortunately, a facility to solve this was introduced with the *CRefactoring2* class, namely the *RefactoringASTCache*. Loading an *AST* using this cache will always return the same instance of the syntax tree if it was already loaded before or create a new instance instead. It is noteworthy to mention that the translation unit loading must be done manually by using the *findElement(..)* method on the associated project. However, it is important to load all translation units using either absolute or relative paths. Creating a translation unit by using an absolute and a relative path is possible, but the cache will create two *AST* instances for these translation units.

F.5 Visitors

Visitors [Gam00] are widely used in refactoring plug-ins. They ease the traversal of the *AST* by visiting specific nodes and allowing to perform custom actions for them. For example,

a visitor implementation can search the selected node in an *AST*. To implement a custom visitor, the abstract base class *ASTVisitor* is available in Eclipse. Applying a visitor to an *AST* can be accomplished by invoking the *accept(..)* method of an *AST* node.

F.6 Casting Nodes

The inheritance hierarchy of a node in the syntax tree can include several different base interfaces. However, methods invoked on node instances and other lookup methods mostly have the highest possible interface in the hierarchy as their return type. Unfortunately, in most situation a downcast is necessary to access required information. This can yield unexpected results. For example, if you try to downcast a node to a given interface, but this type is not implementing this interface, a class-cast exception is thrown. Therefore, you have to add checks to make sure the correct cast is applied. To avoid this, every *AST* node provides a *getAdapter(..)* method. This method is a "safe cast", providing you with the desired instance or null, if the cast would not be successful. However, you still have to check for null values this way.

References

[Ecl11] Eclipse, Foundation: Eclipse CDT Project (2011), URL http://www.eclipse.org/cdt/, timestamp: 2011.11.17

[Ecl12a] Eclipse, Foundation: Eclipse CDT Indigo 8.0.0 Download (2012), URL http://www.eclipse.org/cdt/downloads.php, timestamp: 2012.02.09

[Ecl12b] Eclipse, Foundation: Eclipse CDT Indigo 8.0.0 Update-Site (2012), URL http://download.eclipse.org/tools/cdt/releases/indigo/, timestamp: 2012.02.09

[Ecl12c] Eclipse, Foundation: Eclipse CDT Static Code Analyser (2012), URL http://wiki.eclipse.org/CDT/designs/StaticAnalysis, timestamp: 2012.02.09

[Ema06] Emanuel Graf, Leo Buettiker: *C++ Refactoring Support fuer Eclipse CDT*, Diploma thesis, University of Applied Sciences Rapperswil, HSR (2006)

[Fel11] Felber, Lukas: Static Include Analysis for Eclipse CDT (2011), URL http://www.includator.ch/, timestamp: 2011.11.17

[Fow04] Fowler, Martin: *Refactoring - Improving the Design of Existing Code*, Addison-Wesley (14th Printing, 2004)

[Gam00] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (21st Printing, 2000)

[Ind10] Indermuehle Matthias, Knoepfel Roger: *CDT C++ Refactorings*, Bachelor thesis, University of Applied Sciences Rapperswil, HSR (2010)

[ISO11] ISO/IEC: Working Draft, Standard for Programming Language C++ (2011), URL http://open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf, timestamp: 2011.12.12

[Ker04] Kerievsky, Joshua: *Refactoring to Patterns*, Addison-Wesley (1st Printing, 2004)

[Kes10] Kesseli, Pascal: *Loop Analysis and Transformation towards STL Algorithms*, Master-thesis, University of Applied Sciences Rapperswil, HSR (2010)

[Lar07] Larman, Craig: *Applying UML and Patterns*, Prentice Hall (9th Printing, 2007)

[Mat10a] Mattias Holmqvist: Building with Tycho, Tutorial Part 1 (2010), URL http://mattiasholmqvist.se/2010/02/building-with-tycho-part-1-osgi-bundles/, timestamp: 2012.02.09

[Mat10b] Mattias Holmqvist: Building with Tycho, Tutorial Part 2 (2010), URL http://mattiasholmqvist.se/2010/03/building-with-tycho-part-2-rcp-applications/, timestamp: 2012.02.09

[Mat10c] Mattias Holmqvist: Building with Tycho, Tutorial Part 3 (2010), URL http://mattiasholmqvist.se/2010/06/building-with-tycho-part-3-testing\-code-coverage-and-easier-development-using-target-definitions/, timestamp: 2012.02.09

[Opd92] Opdyke, William F.: Refactoring Object-Oriented Frameworks (1992), URL www.laputan.org/pub/papers/opdyke-thesis.pdf, timestamp: 2011.11.17

[Pro12] Project, Apache Maven: Apache Maven Project (2012), URL http://maven.apache.org/, timestamp: 2012.02.09

[Sch10] Schwab Martin, Kallenberg Thomas: *One touch C++ code automation for Eclipse CDT*, Semester-thesis, University of Applied Sciences Rapperswil, HSR (2010)

[Son10] Sonatype: Maven Eclipse Plugin (2010), URL http://eclipse.org/m2e/, timestamp: 2012.02.09

[Son12] Sonatype, SAP: Tycho Maven Plug-In (2012), URL http://www.eclipse.org/tycho/, timestamp: 2012.02.09

[Thr10] Thrier, Yves: *CloneWar - Refactoring/Transformation in CDT: Extract Template Parameter*, Semester-thesis, University of Applied Sciences Rapperswil, HSR (2010)

[Thr11] Thrier, Yves: *Troi - C++ Template Concepts in Eclipse*, Semester-thesis, University of Applied Sciences Rapperswil, HSR (2011)

List of Abbreviations

ADL	Argument Dependent Lookup
AST	Abstract Syntax Tree
CDT	C/C++ Development Tooling
ECTS	European Credit Transfer System
HSR	University of Applied Sciences Rapperswil
IDE	Integrated Development Environment
TDD	Test-Driven Development

List of Figures