Making Software Refactorings Safer

Anna Maria Eilertsen

Department of Informatics
University of Bergen

Master Thesis

June 2016

Supervisors

Volker Stolz, Bergen University College
Anya Helene Bagge, University of Bergen

Abstract

Refactorings often require that non-trivial semantic correctness conditions are met. IDEs such as Eclipse's Java Development Tools rely on simpler, static precondition checks for refactorings. This leads to the phenomenon that a seemingly innocuous refactoring can change the behavior of the program. In this thesis we demonstrate our technique of introducing runtime checks of two particular refactorings for the Java programming language: Extract And Move Method, and Extract Local Variable. These checks can, in combination with unit tests, detect changed behavior and identify the refactoring step that introduced it.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 Contribution . . . 6
  - 1.2.1 Motivating example . . . 6
  - 1.2.2 Organization . . . 8

2 Background **10**
- 2.1 Programming language preliminaries . . . 10
- 2.2 Terminology . . . 11
  - 2.2.1 Static Analysis . . . 13
  - 2.2.2 Program processing . . . 13
  - 2.2.3 Refactoring tools . . . 13
- 2.3 Refactorings . . . 14
  - 2.3.1 Classification of refactorings . . . 15
  - 2.3.2 Specifying refactorings . . . 16
  - 2.3.3 Preserving Behavior . . . 18
  - 2.3.4 Implementing tools . . . 19
  - 2.3.5 Software Quality . . . 19
- 2.4 Extract Local Variable . . . 20
- 2.5 Extract Method . . . 22
- 2.6 Move Method . . . 23
- 2.7 Extract And Move Method . . . 27

3 The Eclipse Project **31**
- 3.1 The Eclipse SDK . . . 31
- 3.2 The JDT . . . 32
  - 3.2.1 The Java model . . . 33
  - 3.2.2 The Eclipse Java AST . . . 34
  - 3.2.3 Parsing a Java code file in Eclipse . . . 36
- 3.3 Eclipse's implementation of refactorings . . . 37
  - 3.3.1 Language-Independent Infrastructure . . . 37
  - 3.3.2 Extract Method implementation . . . 38
  - 3.3.3 Move Method implementation . . . 39
  - 3.3.4 Extract Local Variable implementation . . . 39

4 Proposed solution **41**
4.1 Extract Local Variable . . . 42
4.1.1 The precondition . . . 42
4.1.2 Expressing the property . . . 45
4.1.3 Location of assert insertion . . . 46
4.2 Extract And Move Method . . . 49
4.2.1 Similarities to Extract Local Variable . . . 49
4.2.2 Generalizing the property . . . 50
4.2.3 Integration into a refactoring tool . . . 51

5 Implementation Details **53**
5.1 The Extract And Move Method Plug-in . . . 53
5.2 The Safer Refactoring Plug-in Overview . . . 55
5.2.1 Infrastructure . . . 56
5.2.2 Development software . . . 56
5.3 Analyzer . . . 56
5.3.1 The Target Argument . . . 57
5.3.2 The Selection Argument . . . 59
5.4 Refactoring . . . 60
5.5 Extract And Move Method . . . 61
5.5.1 Implementing assert generation . . . 62
5.6 Extract Local Variable . . . 65
5.6.1 Implementing assert generation . . . 65

6 Experiment **68**
6.1 Experiment design . . . 68
6.1.1 The test case . . . 69
6.1.2 Set-up . . . 70
6.2 Results . . . 70
6.2.1 Extract Local Variable . . . 70
6.2.2 Extract And Move Method . . . 71
6.3 Discussion . . . 72
6.4 Experiment Conclusion . . . 72

7 Discussion **73**
7.1 Topic motivation . . . 73
7.1.1 Specification discussion . . . 75
7.2 Experiment Discussion . . . 75
7.2.1 Experiment design . . . 75
7.2.2 Threats to validity. . . . 76

8 Conclusion **79**

Appendix A Code examples **80**
A.1 Behavior change code examples . . . 80
A.1.1 Assignment and reassignment in one statement . . . 80
A.1.2 Extract And Move Method with null-target . . . 82

Acknowledgements

First and foremost, thank you Volker Stolz for supervising me through this thesis. Thank you Anya Bagge, for providing me with incredible opportunities. Thank you Magne Haveraaen and Jurgen Vinju for patiently helping me navigate my own thoughts. Thank you Mari Garaas Loechen for doing so much more than your job. Thank you Eivind Jahren for your never ending enthusiasm and support. Thank you Tanya for your effort in editing this thesis. Thank you Haakon, for never backing out of any kind of discussions, no matter how technical. Thank you fellow master students and friends, for sharing experiences, typesetting tips, feedback and frustrations – and some really good times.

Chapter 1

Introduction

In this chapter we introduce refactorings, define our research questions and outline the rest of this thesis.

1.1 Motivation

According to the definition provided by Fowler [10, p 53] a *refactoring* is "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". *To refactor* is the act of making such a change.

Programmers refactor frequently [32]. The popularity of the programming methodologies Agile Programming [19] and eXtreme Programming [3] have aided this development, as they emphasise the importance of continuous refactoring in software development. Small refactorings like `Rename` [10] and `Extract Local Variable` (defined in Section 2.4) should be performed regularly by the developer, to keep the code clean and avoid technical debt [10, 14]. Complex refactorings like `Extract Class`, `Replace Delegation with Inheritance`, or `Convert Procedural Design to Objects` [10] are commonly performed as a free-standing activity, when introduction of new functionality requires a change in architecture, or to decrease the maintenance cost of software by tackling built-up technical debt – possibly as part of a search for bugs. Less complex ones are performed more frequently [30] and can be composed into a larger ones [17].

Refactoring correctly by hand is hard and can introduce errors [22]. To support the ideal practice of frequent refactoring, a number of automated refactoring tools have been developed and common refactorings are often supported by IDEs (integrated development environment). Unfortunately the behavior of automated refactoring tools are not always well documented, and the implementations can differ from what programmers expect. Refactoring preconditions are not always communicated well, neither in the documentation nor in error messages. Most tools have basic precondition checks in place, with typically a minimum at a syntactical check of the input code. These checks are not necessarily sufficient, and may result in refactorings being applied in cases where preconditions are violated. The implementation of the refactoring can also be surprising at times, and, as we will soon see in examples, can deviate from what the programmer intended when she invoked it.

Tool developers have few sources for definitions of refactoring, with Fowler's catalogue of 72 refactorings [10] being the most notable one, along with the implementation of other refactoring tools [9]. The first industrial-grade refactoring tool, the Smalltalk Refactoring Browser [24], inspired the development of a myriad of other refactoring tools for different languages over the next years [9]. The Smalltalk Browser was developed based on Opdyke's catalogue of precondition-based refactoring for object-oriented systems, the first large academic work on refactorings, and one that still – through the Smalltalk Browser and its successors – influences tools today.

A developer that wishes to familiarize herself with the refactorings supported by tools has the options to play around with the tool, or to read the available literature on refactorings and then assume that the tool implements them in the described manner. Taking into account the variety of programming languages the scientific literature falls short in describing refactorings, and eager developers will resort to Fowler's catalogue – later extended with Kerievsky's Refactoring to Patterns [14] – or to refactoring descriptions found online. These catalogues are much too ambiguous to serve as specification for toolmakers, which must implement their refactorings ad-hoc. Thus – understandably – the tools vary their implementations and precondition checks, which in turn makes it hard for the programmer to grow confident in her use of these tools.

Currently, the commonly suggested "solution" by Fowler, among others, is: never refactor without a proper test suite. This correlates with the authors' dependence on the developer's own judgement in how to implement the described refactoring in the particular cases. This leads to descriptions that are hard to implement in tools, which in turn leads to ad-hoc implementations that can surprise the programmer, break the code, and introduce subtle changes in the semantics of the code that will not show up as a compilation error, and it requires carefully crafted tests to reveal them. Some of these errors are hard to identify manually, and requires a solid programming language understanding of programmer [29]. Such problems prevents the adoption of tools, and the need for more correct refactoring tools has been voiced several times [4, 9, 15]. At the same time, making the tools too restrictive or rejecting refactoring invocations when the programmer does not understand why it is prevented will also inhibit use, and can make the programmer simply perform the same (possibly unsafe) refactoring by hand [4, 9, 15, 30, 21].

Although proper tests should undeniably be in place, we would like to address the semantic precondition checking. While syntactic preconditions can effectively be checked statically, semantic preconditions can pose problems in languages like Java where doing determining whether a path is executed can be undecidable by static analysis [1].

Research question

Can refactorings be made safer by encoding preconditions as dynamic checks?

We consider one semantic precondition that is hard to check statically and we propose a dynamic check as an alternative. More specifically we aim to encode in the source code a semantic property of a program structure that predicts changed behavior after refactoring, and we introduce a runtime check in the code that will alert the programmer if the structure does not have this

property. In our experiment section we illustrate how this, together with a well-covering (although not necessarily complete) test suite can check correctness of two refactorings in Java: `Extract Local Variable` and the composite refactoring `Extract And Move Method` [17]. We validate that our idea would "work in the wild" with a case study, in which we automatically apply a high number of the enhanced refactorings to a commonly used, open-source software project and check the introduced preconditions dynamically to see if they detect changed behavior. This is a considerable step in the direction of a safer industrial strength refactoring tool, that currently has a safer `Extract Local Variable` refactoring ready-for-use with a safer `Extract And Move Method` close behind.

The core content of this thesis is accepted for publication at the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016) [7]. This thesis builds on the work by Kristiansen [17] described in Section **??** and for our experiment we reuse parts of his plug-in, described in Section 5.1.

1.2 Contribution

In this thesis we consider the refactorings `Extract Local Variable` and `Extract And Move Method`, and how their correctness can be improved in the Java programming language by generating dynamic checks in the form of assert statements. We describe the development of our proof-of-concept Eclipse Plug-in: Safer Refactoring Plug-in, which provides a drop-in replacement of Eclipse's two refactorings, and report our findings from our case study on whether such corner cases of Java code can be found in the wild, on the Eclipse JDT UI project.

All code examples and refactoring definitions, as well as our implementation, is in and for the Java programming language.

1.2.1 Motivating example

We will give a motivating example of the problem we try to tackle, and show our solution.

Consider the code fragment in Listing 1: a method is repeatedly invoked on the field `x`, which is assigned to between the two method invocations. Our example is simplistic, but in a large code base such behavior may not be as evident, and the structure of our example can be generalized to longer sequences of statements, as discussed in Chapter 4. APIs (Application Programming Interface) frequently require sequences of invocations, and to avoid repetition a programmer may decide to refactor this sequence into a new method in the target class or extract a complex, repeated expression into a local variable. In order to assure behavior preservation in this case, the tool must check that the expression is not assigned to in the refactored code. If the assignment is a statement in the refactored method body, this refactoring would be prevented by most tools, but checking a precondition stating that an assignment cannot happen in any code reachable from the method is hard, and often impossible, in Java. To avoid posing too restrictive preconditions, like preventing the refactoring if an assignment cannot be ruled out, this refactoring will be executed, producing well-formed code with no compile errors, but with a subtle change

in semantic: whereas the calls before have been on distinct objects, they are now on a single object. This may or may not be a problem, depending on how the program behavior is defined. It does however, mean that the refactoring can change program behavior without notifying the programmer, which is undesirable in a tool. The same effect can be observed with the refactoring `Move Method`, as we explain in Section 2.6 and show in Listing 8.

In the popular Java IDEs Eclipse and NetBeans, performing `Move Method` on the original example provided in Listing 1 will also yield this kind of behavior change, while a different Java editor, IntelliJ, will use another implementation of the refactoring with its own problems (discussed in Section 2.3).

Original code

```
1  class C {
2    X x = new X();
3
4    public void f() {
5      x.n();
6      m();
7      x.n();
8    }
9    void m(){x = new X();}
10 }
```

After Extract Local

```
1  class C {
2    X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      m();
8      temp.n(); //semantic change
9    }
10   void m(){x = new X();}
11 }
```

Listing 1: Example of a behavior-changing application of `Extract Local Variable`. Extracted expression is highlighted in original code and replaced throughout the whole scope. Due to an assignment in another scope this behavior-changing refactoring is allowed by two of the most common Java IDEs, Eclipse and NetBeans.

We claim that the crucial behavior change happens at the point in the code where the reference to x is replaced with a reference to *temp*, which happens under the assumption that the two are behaviorally equivalent, and we propose an encoding of this assumption: `x == temp`. We can check this encoded property dynamically using the `assert` keyword in Java. Adding generation of this dynamic check to the `Extract Local Variable` refactoring will produce the code shown in Listing 2.

We will show that this property is a necessary precondition for both `Extract Local Variable` and `Extract And Move Method` (a particular instance of `Move Method`). For `Extract Local Variable` this will act as an *postcondition*, i.e. a condition that must be true after the refactoring if it was applied correctly; while for `Move Method` and `Extract And Move Method` they can either be use in the same way, or as runtime preconditions. If the code shown in Listing 2 runs without the asserts alerting the programmer, then the same precondition holds for `Extract And Move Method` and `Move Method`. The programmer can apply `Inline Local Variable`, remove the assert (ideally automatically by the tool) and safely apply the more complex refactoring. This is described in-depth in Chapter 4.

In order to run the checks the code containing the asserts must be executed, either by manually running the program or by running a test suite guaranteed

Refactored source

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      m();
8      assert x == temp: "Extract Local changed semantic";
9      temp.n(); //semantic change
10   }
11   void m(){x = new X();}
```

Listing 2: The result of applying `Extract Local Variable` with dynamic checks to the code in Listing 1: dynamic checks are added and will alert the programmer at runtime that the objects change.

to execute these lines of code. Note that the test suite does not need to actually test the result of running the program, and not even to exhaust the execution path, as long as it *ensures coverage of all execution paths that can lead to the refactored code*. This reduces the effort of writing the tests significantly.

Options for implementation of the dynamic checks, as well as a general discussion on the semantic change, are covered in Chapter 4, with extensive examples found in Appendix A.

1.2.2 Organization

Here we describe the different chapters:

- In Chapter 2 we investigate different interpretations of the refactorings `Extract Local Variable` and `Extract And Move Method` and define algorithms for the ones we use. We visit related works and define the terminology and the programming language preliminaries used throughout this work.
- In Chapter 3 we give the background needed on Eclipse.
- In Chapter 4 we look in-depth at the problem the refactorings have in relation to behavior preservation and why the dynamic checks we propose are in fact a solution to the hard-to-check precondition. We consider different ways to determine where to insert the checks, and different ways they can be represented in Java.
- In Chapter 5 we describe how we implemented our ideas from Chapter 4 and the general development of the plug-in for our experiment. We also describe the plug-in we took as starting point for this thesis.
- In Chapter 6 we describe our experiment setup, implementation and findings, and discuss the experiment design.
- In Chapter 7 we discuss our idea, our implementation choices, our choice of experiment and approach to validation, and list future work.

- In Chapter 8 we summarize and conclude this work.
- In the Appendix we look in-depth at some problems described briefly in this thesis, list some bugs in Eclipse, and give more extensive code for some examples given in Listings, that would otherwise make this document too large.

Chapter 2

Background

In this chapter we will define the terminology used in this thesis, give an introduction to refactorings and the related literature and tool support, as well as the commonly used definitions of the refactorings we will consider. We describe terminology and program processing using the minimum level of detail that enable us to refer to it throughout this document and should not be considered an attempt at an exhaustive explanation.

2.1 Programming language preliminaries

A programming language is a language that can be translated into, or *compiled* to, machine commands. Such languages are used to write computer programs.

The *syntax* of a language defines the basic components (also called *tokens*) allowed in the language, and how they can be combined structurally. The *semantics* of a language is the meaning encoded in these constructions. *Static semantics* is the requirements a syntactically correct program must fulfill to be considered "well-formed", or executable. *Dynamic semantics* or "execution semantics" is the behavior of the program at run time.

Correctness of programs can be discussed in terms of syntactical correctness: whether a program is well-formed, i.e. conforms to the syntactic rules of its language; static semantics: whether the the program has structural or type errors; and dynamic semantics: that the executed program conforms to its specification. In Java, type checking is part of the static semantics, while software testing is typically done on the level of dynamic semantics.

Static checking is relevant for instance in IDEs to produce compilation error messages or warnings, like syntax errors, type errors, dead code and unused imports. In the static semantics of Java, we consider the *static type* of the program elements, i.e. their declared type. For object references this does not have to be the same type as their *dynamic type*, which means that in a static analysis we cannot always know which in which type declaration a member will be looked up.

2.2 Terminology

We will assume that the reader is familiar with the commonly used Java programming language syntax and terminology, although we do explain some of the core Java language constructs we use in the following list. We will use the following terminology in this work:

Expression: A programming language element that can be evaluated to a single value. The value can have type `void`.

Statement: A programming language element that forms a complete unit of execution. In Java statements are marked by an ending `;`.

Selection: An ordered list of consecutive program statements. When a selection is not a list of statements, but rather one expression, we will refer to it as a selected expression.

Well-formed: Conforming to some rules. The term is often used in conjunction with selection. A *well-formed selection* conform to different set of rules depending on the context, for example it can be required to have no return statement, to have return statements at all execution paths, or simply be syntactically correct. The conditions under which a selection is well-formed will be described as is needed.

Assertion: In Java this is a statement that tests assumptions about the program. It has two forms:

- `assert expr1;`
- `assert expr1 : expr2;`

where `expr1` evaluates to a boolean value, and `expr2` is a non-void expression. If `expr1` evaluates to `false` at runtime the program will halt with a `AssertionError`; if the `expr2` has been provided this will be the error message of the AssertionError. Evaluation of asserts can be turned off and on and they are not evaluated by the virtual machine by default. When disabled it is equivalent to an empty statement, and none of the arguments are evaluated. They are enabled by providing the argument `-ea` to the virtual machine.

Final: A Java modifier which, when added to a variable declaration, assures that it is only assigned to once, i.e. cannot be reassigned. A final field must be assigned to in all constructors or initializers in the class it is declared in.

Reflection API: A feature in Java that allows runtime lookup and modification of code.

Set-method: A method whose only purpose is to set the value of a field. Example in Listing 3.

```
1 public class C {
2     X x;
3
4     public void setX(X x){this.x = x;}
5     public X getX(){return x;}
6 }
```

Listing 3: Typical example of set-method and get-method.

Get-method: A method whose only purpose is to get the value of a field. Example in Listing 3.

Navigation path: Dot-separated names, e.g.: `this` in `this.m();`, `x.y.z` in `x.y.z.m();`

Qualifier: A navigation path prefixing members, specifying the reference or type in which the member should be looked up.

Prefix: A term borrowed from Kristiansen [17], where it is defined as a navigation path prefixing a member or on its own. We extended it to possibly include get-methods.

Unfix: A term borrowed from Kristiansen [17], where it is defined as a Prefix that is unfit to be a target of the `Extract And Move Method` refactoring, usually because it is assigned to in the scope.

Shadowing Declaring a variable with a name already present in the namespace. Example: `class C{X x; foo(){X x;}}`

Namespace: The collection of all names defined in a scope, i.e. all names visible from a particular place in the code.

Fresh: A fresh name to a namespace is a name that does not already exist, or have existed, in the namespace.

Regular expressions: A sequence of characters that define a search pattern. We will use the following operators known from regular expressions, with their corresponding definitions:

- `X*` means zero or more `X`s
- `X+` means one or more `X`s
- `X?` means zero or one `X`s

2.2.1 Static Analysis

Static analysis is program analysis performed without running the program. Such an analysis is performed on the statically available information about the program (control flow, data flow, internal data) and utilises the dynamic semantic to predict runtime behavior. FindBugs[1] is one tool making use of static analysis. It is a tool for finding bugs in Java code, and software for calculating metric scores. The technique can be utilized for *pointer analysis* or *reference analysis*, a type of program analysis that aims to establish what program pointers can refer to throughout program execution [25]. As such, limitation in static analysis of a language will propagate to limit the completeness of the results such techniques can produce statically: advanced static analysis tools can help determine possible execution paths in the program and can help with excluding impossible scenarios, but in general the presence of Input/Output, random generator, the reflection API and external libraries, make it impossible to guarantee a complete execution graph of Java programs by the means of static analysis.

2.2.2 Program processing

Computer programs are usually written to, and stored in, a text file as source code. A compiler is a program that can turn a textual representation of a computer program into an executable program. In this representation they have no explicit structure. During the compilation process a *parser* turns a *tokenized* ("cut" into tokens) version of the source code into a structured model of the code. There are different paradigms for which models to use: a common one, and the one Eclipse uses, is the AST (*abstract syntax tree*). An AST is a tree structure where the program elements and structure are encoded as nodes and edges. The AST encodes just enough information to preserve the meaning of the original code, and abstracts away superfluous syntax, like keywords and punctuation. The AST can be turned back into source code by *pretty-printing* or it can be further compiled into machine code. This is illustrated in Figure 2.1.

For an example of an AST, consider the expression `int i = 2 + 1;`. This could have the AST defined in Figure 2.2. The order the operators should be evaluated in is encoded in the structure of the tree, and the semicolon is implicit from the root node being a `Statement`.

The syntax of ASTs varies, and can be defined in a regular tree grammar. The grammar itself can also be thought of as a tree (or a graph, if it is recursive).

2.2.3 Refactoring tools

Refactoring tools provide an interface for the programmer through which she can refactor without editing text directly. Tools usually operates on the intermediate model of the source code, for instance the the AST.

This is illustrated in Figure 2.3 with a typical use case deciphered in Figure 2.4. The user can edit source code directly, or she can invoke refactorings through the tool interface. The tool will load and analyze the model of the source code – rarely does the refactoring tools provide their own parser, and the changes are usually too complex to be performed on the source code – and provide feedback to the user concerning preconditions, warnings or previews. It

[1] http://findbugs.sourceforge.net/

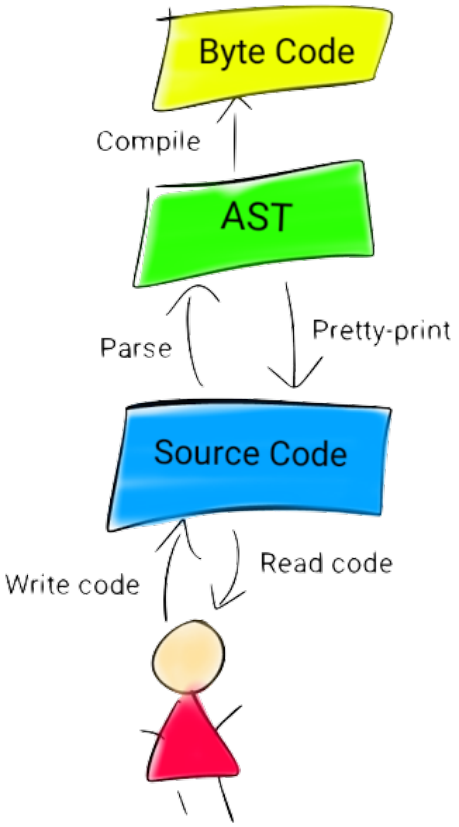

Figure 2.1: Program processing in Java (simplified): the developer operates directly on the source code by reading (*comprehending*) and editing the textual representation of the program through a text or code editor, like Eclipse (described in Section 3.1). An intermediate, abstract representation by a parser, which can be pretty-printed as source code, or compiled into an executable program.

performs the refactoring as a model- or tree transformation, defining the changes that should be performed, and can leave it to the pretty-printer to turn them into textual changes. Finally the source code is updated (as well as the byte code, if the program is compiled) and the user can see the results.

2.3 Refactorings

In Chapter 1 we gave the commonly used definition of refactoring and described refactorings in the context of software development and program functionality. We described some of the problems that makers and users of refactoring tools face, and gave an example of applications of refactorings that can change the program's semantic in the most common Java IDEs. In this section we will list related works on refactoring specification and tools, and discuss what it

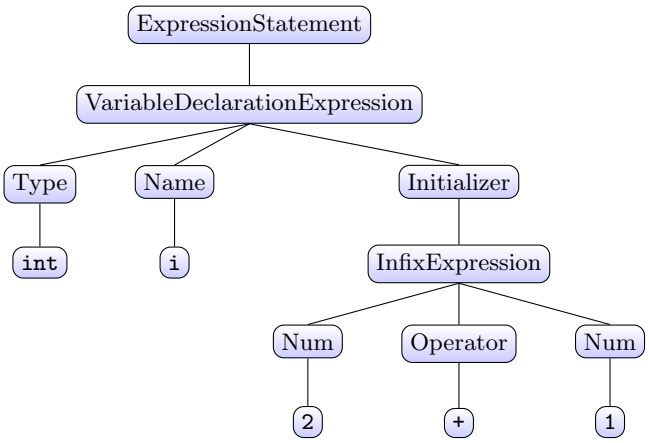

Figure 2.2: Example of an AST loosely corresponding to Eclipse's JDT AST, described in Section 3.2.2. Here we have simplified the node types and names.

means that a refactoring is behavior-preserving and how it can be assured. In the next sections we will aim to define the refactorings we us in the rest of this thesis: `Extract Local Variable` in Section 2.4, `Extract Method` in Section 2.5, `Move Method` in Secton 2.6 and their composition `Extract And Move Method` in Section 2.7. We will define them for the Java programming language, and we will give a lower bound of the definition, with assumptions that make them simpler to define while still working in our examples.

2.3.1 Classification of refactorings

It is common to distinguish refactorings of different complexity by referring to them as low-, medium- or high-level refactorings [23, 30, 15, 22, 9]. As the source code changes become more widespread and requires increasingly sophisticated machinery to determine the textual changes needed, the refactorings increase in level. This classification is not absolute or discrete, but it provides a useful terminology and indicates both the likeliness of a refactoring being performed [30] and, as we will see in the following sections, the difficulty of implementing it in a tool.

- A *low-level* refactoring is a local change, often confined to one scope as in the case of `Rename Local Variable` or `Inline Local Variable`, and can sometimes be solved with lexical operations.
- A *medium-level* refactoring – although sometimes merged with high-level – will affect a larger part of the code, like a whole compilation unit, and require complex syntactical or semantic analysis, as in the case of `Extract Method`, `Rename Method` and `Move Static Member`.
- A *high-level* refactoring is a global change that can affect the whole workspace and require complex reasoning, like `Extract Class`. High-level refactorings can be composed of two or more low- and medium-level refactorings, as in the case of `Extract And Move Method`, but some high-level

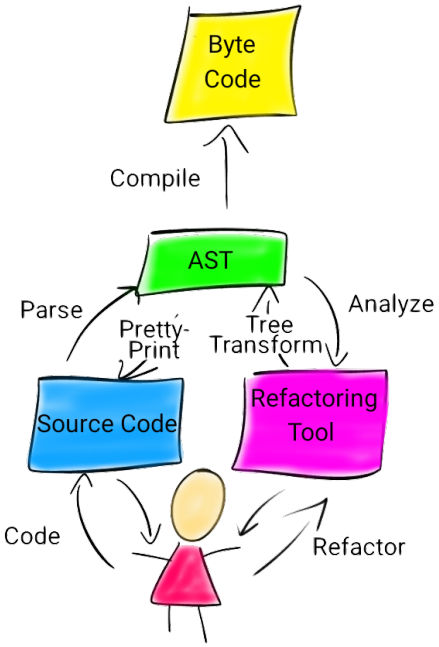

Figure 2.3: A Refactoring tool shown in relation to program processing: the user can edit the program both through the source code, and through the refactoring tool. The tool can analyze the intermediate program model and define transformations of it. If the model is transformed, the source code must be updated, which relies the textual changes to the user. For a use case consider Figure 2.4.

> refactorings, like `Convert Procedural Design To Objects`, are hard to factor into discrete steps.

2.3.2 Specifying refactorings

As mentioned, resources specifying the behavior of refactoring instances are lacking, which pose problems in regards to implementing automated refactorings. Ideally a catalogue of refactorings should exist, formal enough to serve as specifications for the toolmakers, and readable enough to serve as a guide for the developers that use the tools. Catalogues of refactorings have been attempted by Opdyke [23] and Fowler [10].

Opdyke. In his seminal doctoral thesis [23], Opdyke introduced *precondition-based refactoring*: refactorings that are behavior-preserving given that their listed preconditions holds. He defined 26 low-level refactorings for C++ and composed them into a number of high-level, behavior-preserving refactorings that could be useful in any object-oriented system. Opdyke defined his refactorings such that he could argue correctness when preconditions were upheld. This work was later used to implement the first industrial grade automated refactoring tool [9], the Smalltalk Refactoring Browser [24], which over the next

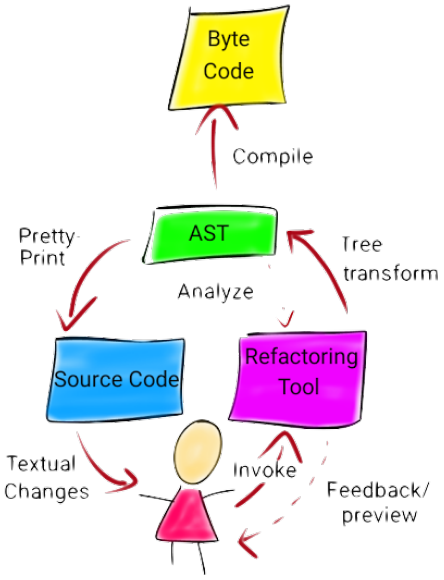

Figure 2.4: The user invokes a refactoring through the interface of the tool. The tool loads and analyzes the current AST, and provides feedback like warnings and errors, or asks the user for additional input. Then it compiles the required tree change and transforms the AST. The user sees the change when it is pretty-printed as source code and reloaded in the editor.

two decades inspired a myriad of refactoring tools and plugins for many languages and IDEs [9]. Unfortunately, Opdyke's catalogue is perhaps too specific to serve as specification of the commonly used refactorings today: when we searched his thesis for a definition of `Move Method`, the only refactoring we find that resembles what we think about as `Move Method` is `Moving Members into a Component` and `Moving Members into Aggregate`, where neither behaves like the refactoring offered by the Eclipse and NetBeans. These examples are further discussed in Section 2.6. Even though his theoretical foundation is still actively consulted in the academic literature, his catalogue does not pose a solution to the problem of specifying refactorings.

Fowler. Later a catalogue of 72 language-independent refactorings for object-oriented programs was published by Fowler [10]. His book was followed by Kerievsky's book [14] which described refactoring practice to and from design patterns, leaning on Fowler's catalogue. Fowler took the approach of defining refactorings not as a formal specification, but rather as a sequence of natural language instructions to the human programmer. He aims to illustrate, through UML-diagrams, code examples and informal descriptions, when and how the programmer should refactor, as well as attempting to help the programmer go through with the code change without breaking existing functionality. He relies heavily on the programmers ability to interpret his instructions the right way, as illustrated in one of the steps in`Move Method`:

"Copy the code from the source method to the target. Adjust the method to

make it work in its new home. (...) If the method includes exception handlers, decide which class should logically handle the exception."

It is often left to the reader to figure out the implementation details, and Fowler stresses the importance of writing tests before refactoring, as the refactorings in the book can, and will, change behavior in intermittent stages before they are considered finished. As such, his refactorings are not guaranteed to be behavior-preserving, and relies on the programmer's abilities and test suites. The popularity of his book makes it relevant as a measure of what refactoring tools should offer. Indeed, Fowler has named what is known as the *refactoring rubicon* – a minimum that should be offered by a refactoring tool – as a correct implementation of `Extract Method`. This is further discussed in Section 2.5. However, his catalogue is not complete, and does not serve as unambigous specifications.

2.3.3 Preserving Behavior

The most commonly used definition of refactoring is the one provided by Fowler, as given in Section 1.1. Present in his and other (like Opdyke's [23]) definitions is the mention of preserving behavior. Fowler says the refactoring must not change the program's "observable behavior", and Opdyke elaborated on this property [23, p 2]:

"Refactorings do not change the behavior of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same. Refactorings are behavior preserving so that, when their preconditions are met, they do not "break" the program."

Current tool support does not adhere to these definitions [9] even though behaviour-preservance is considered of key importance in a refactoring tool and lack thereof can lead to avoidance of the tool [4]. However, a proper definition of behavior is lacking, and thus also of how to preserve it. Even Opdyke's definition using "resulting set of output values" is be open to interpretation: should the call graph be preserved, or string output? What about graphics? Does subtle changes in behavior that currently is not affecting output, but might affect future output count? What about computational time and debugging output? Recent research [15] has cast light on the discrepancy between the rigid definition of refactoring found in literature and what developers mean. This is a difficult discussion, voiced by notable voices like Steinmann and Brant [4, 9, 15], and not one we try to solve here. We proceed without a universal, unambiguous definition of behavior, and we keep in mind the loose (or lacking) definition when we discuss behavior change and precondition checking in Chapter 4 and in the discussion in Chapter 7.

In this thesis we assume that the semantic change shown in 1 can result in a behavior change if the behavior of the program depends on the objects the method is called on – a common occurrence in object-oriented programs. We further assume that the programmer knows best what kinds of behavior change are acceptable, and that an important part of the refactoring tool's responsibility is notifying the programmer in case of such behavior change, not necessarily to always refuse it.

2.3.4 Implementing tools

Advancements in automated refactorings is aimed at providing programmers with tools that will enable them to refactor faster and more securely, as a natural part of their work flow. The performance of these tools is vital: programmers who report underuse of refactorings mention reliability and correctness of the tools among the reasons [30, 15, 4]. Therefore it is concerning that many commonly used IDEs with refactoring support have a high number of reported bugs [6] that can require in-depth understanding of the software language and program specification to be manually detected. Although refactoring is recommended to be done in combination with a solid set of tests, tools should ideally guarantee a certain level of behavior-preservance – or at least notify the programmer when the semantic has been changed.

Implementing refactorings and precondition checking correctly is hard, especially in a language like Java with complex and rapidly changing functionality that does not lend itself well to static analysis. To preserve behavior preconditions must be defined and checked, with violations communicated to the user. Some preconditions are impossible to ensure by static analysis in Java, and tool developers are left with the following choices:

- make overly restrictive tools that prevents the user from executing refactorings that might be correct.
- make tools inadequately restrictive tools that allows the user to execute refactorings that might be incorrect.
- to find other, non-static solutions to precondition checking and thereby ensure correctness.

Too restrictive tools that rejects refactorings the user knows to be behavior-preserving, or does not understand why is rejected, is, as we have seen, one of the reasons developers avoid using tools and – by extension – avoid refactoring [30, 4]. However, trusting the programmer and the test suite to determine which refactorings can be safely executed leads to lack of trust in the tool, which again makes programmers avoid using it [4, 15]. In experiment settings it has been shown that developers can have a low understanding of the implications of a refactoring and the preconditions posed on the source code, and that warnings or previews can sometimes be ignored [30], leading to *unsafe* refactorings.

Currently, implementations in industrial-grade refactoring tools are ad-hoc, with different interpretations of the same refactorings. For Java, the most notable tools are Eclipse, IntelliJ and NetBeans, all of which offer a well-furnished refactoring menu, but lacks the proper precondition testing. An inspection of the Eclipse bug tracker reveals numerous cases of refactorings producing code that no longer compiles correctly. [2] An overview of the current tool support is provided in the most recent survey paper [9].

2.3.5 Software Quality

Refactoring can improve software quality [10, 9]. We will list common code quality benefits of applying the refactorings we will look at, and consider them

[2] See the https://bugs.eclipse.org/bugs/buglist.cgi?bug_status=UNCONFIRMED&bug_status=NEW&bug_status=ASSIGNE for approx. 70 outstanding issue on the `Extract Method` refactoring alone.

input : e – an expression of non-void type E
: S – a selection, as a list of consecutive statements
: *context* – the outermost, non-type scope containing S
output: *context* with e extracted to a local variable in S

```
1 v ← fresh variable name;
2 for s ∈ S do
3 |  in s replace all occurrences of e with v;
4 end
5 add a new variable declaration E v = e context just before S;
```

Algorithm 1: `Extract Local Variable` algorithm: add a new variable declaration initialized to the target argument in the beginning of the selection, then replace all occurrences of target with a reference to the variable.

as motivation for a programmer who wish to invoke one such refactoring. This is not an exhaustive list, but one we use as foundation for the heuristic described in Chapter 5.

- `Move Method` correlate to coupling between classes, a metric measuring references between class, and can reduce repetition of qualifiers of method invocations [9].
- `Extract Method` can reduce length of a method and is in some cases performed as a preparation of `Move Method` [10].
- `Extract Local Variable` can reduce the complexity of the code, by extracting a hard-to-read expression to an explaining variable and avoid repetition, and can sometimes be performed as a preparing step for `Move Method` [10].

2.4 Extract Local Variable

`Extract Local Variable` (also called `Extract Variable`, `Extract Temp`, or `Introduce Explaining Variable` [10, p.124]) is a pattern for extracting an expression to a local variable and replace occurrences of the original expression with a reference to the variable. This is a low-level, commonly used refactoring, listed as the second most used refactoring by Eclipse's tool logging [22]. It can be useful on its own, to improve readability or optimize performance, or it can be performed in preparation for e.g. `Extract Method` or `Move Method`. The algorithm we take as a starting point in this work is given in Algorithm 1 followed by an example and a textual definition and discussion of the preconditions.

Example In Listing 4 we see an example of the refactoring: the expression argument is highlighted, and the selection is taken to be the whole method body, as is common practice in tools. The target expression could have been selected from either occurrence. It is a low-level refactoring, affecting only the code inside the scope it is invoked on, and it can seem like it can be implemented solely as a regular expression find-and-replace. Nonetheless, the admittedly intentionally complicated example in Listing 5 shows us that some level of static analysis is

Before

```
1  public void f() {
2    a.b.c.d.m();
3    a.b.c.d.n();
4    a.b.foo(a.b.c.d);
5    a.b.bar();
6    a.b.c.d.m();
7    }
```

After

```
public void f() {
  D temp = a.b.c.d;
  temp.m();
  temp.n();
  a.b.foo(temp);
  a.b.bar();
  temp.m();
}
```

Listing 4: A simple example of Extract Local Variable. The expression argument is highlighted, and the selection is taken to be the whole method body. The target expression could have been selected from either occurrence.

needed to determine the right replacement locations. The same problem can appear when shadowing variables either from a field or by a local class.

Preconditions. These preconditions ensure that the resulting code is well-formed and behaves the same as before:

- The selected expression is of a non-void type.
- The selected expression has no side effects.
- The selected expression or its aliases are not assigned to within the code that is reachable from the selection.
- The selection is not the outermost type declaration in a compilation unit.
- The program is well-formed, i.e. syntactically correct and type-checks (compiles)

Options. Usually the refactoring is offered with the option of replacing all occurrences or only the selected one. We assume that we replace all, but the algorithm can easily be modified to accommodate this option. We also assume the new variable name to be fresh: usually the user will provide a name, and the refactoring must check that is isn't already used, and if so, give a warning in the case of shadowing, or give an error message and abort.

Precondition checking. Note that the precondition concerning reassignment is not limited to checking that the selection does not contain assignment statements. As we see in Listing 6 the reassignment can be hidden in a method call, even in a virtual method, or an external library. If a is public or has a set-method, and a reference to `this` is passed to the method call, then an assignment can be executed anywhere. To confidently exclude such a scenario from happening, a semantic analysis of the whole program is needed, and for Java, that means that statically guaranteeing an application of `Extract Local Variable` to be behavior-preserving is impossible. This is further discussed in Chapter 4.

2.5 Extract Method

`Extract Method` [10, p.110], `Extract Function`, or `Convert Code Segment to Function` [23, p.75] is a pattern for extracting a sequence of one or more statements to a method and replacing occurrences of them with an invocation of the mehod. It is useful to reduce duplication, reduce method size and partition code into conceptually meaningful *chunks*, in particular because it allows the programmer to decide on a meaningful name for his new function. It is classified a medium-level refactoring since it affects only the code in its compilation unit; it is conceptually intuitive, but still requires a careful analysis and manipulation of the parse tree. Fowler uses it for his *Refactoring Rubicon*, as a measurement of the effort put into a refactoring tool. According to Eclipse's user data, it is ranked as the fourth most used refactoring.

Depending on how in-depth we wish to describe the algorithm of `Extract Method`, it can become very large and fastidious. We describe a simple version in Algorithm 2: it is not our intention to provide a complete implementation specification in this work, but rather to describe it in just enough detail that it serves our purpose in the later chapters. We make the following assumptions:

- We only replace the specified selection, not all occurrences.
- The resulting method is public: we don't have to consider visibility if we want to move it (see Section 2.7)
- The new method is declared with a fresh name, to avoid having to look for overloading or -loaded methods, and having to match signatures if that is the case.
- target and source class is the same: we consider moving it to another class as a different refactoring (see Section 2.6) – although this is an option of `Extract Method` in Eclipse.

Preconditions. These preconditions ensure a well-formed program with the same behavior.

- At most one non-field variable (local variable or parameter) can be assigned to in the selection

Before

```
1  public void f() {
2    a.b.a.b.m();
3    a.b.a.b.n();
4    a.b.foo(a.b.a.b);
5    a.b.bar();
6    a.b.a.b.m();
7    }
```

After

```
public void f() {
  B temp = a.b;
  temp.a.b.m();
  temp.a.b.n();
  temp.foo(temp.a.b);
  temp.bar();
  temp.a.b.m();
}
```

Listing 5: `Extract Local Variable` with expression argument highlighted. A textual replace is not enough to produce syntactically correct code this time.

Before

```
class C{
    A a = new A();

  public void f() {
    a.m();
    newA(); // a = new A();
    a.m();
  }

  public void newA(){
      a = new A();
  }
}
```

After

```
class C{
    A a = new A();

    public void f() {
      A temp = a;
      temp.m();
      newA();
      temp.n(); //changed semantics
    }

    public void newA(){
      a = new A();
  }
}
```

Listing 6: `Extract Local Variable` might change the behavior of the program in ways that cannot be guaranteed to be detected statically.

- The selection cannot have any conditional returns.
- The selection can not have branches to code outside the selection, like e.g. `break` or `continue` statements with target outside the selected code.
- The program is well-formed, i.e. syntactically correct and type-checks (compiles)

2.6 Move Method

`Move Method` is a pattern for moving a method member from one type to another and change all its invocations correspondingly. It is useful to reduce the size of a class, reduce coupling and to move responsibilities to a conceptually better fitted place, such as extracting a superclass. It is a high-level refactoring that requires a number of different textual changes depending on the properties of the moved method and the context in the call sites.

In the literature and tools this refactoring is referred to various ways. Tools will often provide the general option `Move`, which comprises `Move Member`, `Move Constant`, `Move Class`, and so on. `Move Member` includes `Move Instance Method` and `Move Static Method`, but although the implementations differ, the UI does not necessarily show them as different options to the user. Fowler [10] simply lists `Move Method` as one refactoring, and leaves it to the user to implement the correct version, while Opdyke [23] takes the different route and considers the more specialized refactorings `Moving Members into a Component` and `Moving Members into Aggregate`. From this myriad of refactorings, we will consider each refactoring not as one algorithm, but rather as a *set of interpretations* represented by their name, or "intent" (like " `Move Instance Method`"), and illustrate their relationships in Figure 2.5. The `Move Method` refactoring we will use is `Move Instance Method`, although we will refer to our refactoring by the name `Move Method`, and in our algorithm we will lean towards the function-

Before

```
class C{
  public void f() {
    A a = new A();
    B b = new B();
    a.m();
    b.n();
    a = new A();
    a.m();
  }
}
```

After

```
class C{
    public void f() {
      A a = new A();
      B b = new B();
      a = mn(a, b);
      a.m();
    }

    A mn(A a, B b){
      a.m();
      b.n();
      return new A();
  }
}
```

Listing 7: A simple example of `Extract Method`. The selection argument is highlighted and replaced with a method call to a method with the selection as its body.

ality it provides in the Eclipse and IntelliJ IDEs. We point out that alternative implementations of the same refactoring have been described by Opdyke, one of which is implemented in the Netbeans IDE. We will discuss their difference and our choice later in this section.

`Move Method` is usually described using the UML diagram in Figure 2.6 along with a natural language explanation [10, p.142]. This type of definition is too limited for our use, and we give a pseudo code algorithm for Java in Algorithm 3. As in the previous section, this is not a complete specification: we aim for just enough detail to allow us to use this as the `Move Method` definition in the rest of this work. To simplify the algorithm we make the following assumptions:

- The method is invoked in exactly one place: in combination with `Extract Method` with "replace only one occurrence" this will hold and allows us to simplify the algorithm significantly. Alternatively it could have been invoked in several or none places.
- The single invocation is not through the reflection API.
- The method's name is unique: in combination with `Extract Method` where a fresh name is generated, this holds, and lets us avoid looking for overloaded or overloading methods in source and target type hierarchy.

Preconditions. These preconditions ensure a well-formed program with the same behavior. There are additional preconditions posed if we do not make the assumptions we did, in particular concerning overloading methods. These are discussed by Opdyke and Fowler [23, 10].

- Target type is not generic: in general it is very hard or impossible to move a method into a generic type

```
   input  : C – the class the method should be extracted from and to
          : S – a selection in C
          : context – the innermost, non-type scope containing S
   output: a syntactically correct class C in which the selection is replaced
           with a call to the new method m
 1 m ← new method with fresh name;
 2 Add m to C;
 3 Make m static if context is static;
 4 V ← all v in S such that v is declared before S and
 5     v is referred to after S and
 6     v is not assigned to in S;
 7     for v in V do
 8         add v as parameter to S with same name and type;
 9         if context inside a method and
10         type of v refers to that method's type argument then
11             add type of v as type parameter of m;
12         end
13     end
14     set body of m to S;
15     if there exists exactly one variable v such that v is assigned to in
       S and
16     v is referred to after S and
17     v declared before or in S then
18         set return type of m to the declared type of v;
19         if v is declared before S then
20             declare it a local variable v2 with name and type of v in the
               start of m;
21             return v2 as the last line of m;
22             replace S in context with an assignment to v where the right
               side is a call to m with the appropriate arguments;
23         end
24         else
25             replace S in context with a declaration of v, with a call to m
               with the appropriate arguments as initializer;
26         end
27     end
28     else
29         set m's return type to void;
30         replace S in context with a call to m with the appropriate
           arguments;
31     end
```

Algorithm 2: `Extract Method` for Java. We assume target and source class to be the same, and to replace only the selected occurrence.

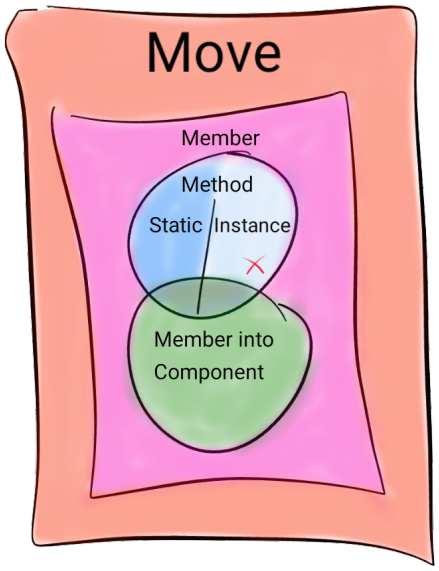

Figure 2.5: `Move` refactoring hierarchy. The outermost set denotes all `Move` refactorings, with all `Move Member` refatorings being a subset, of which *Move Method* is a subset, partitioned into `Move Static Method` and `Move Instance Method`. The green circle denotes Opdyke's `Move Memver into Component`, which intersects with both `Move Static Method` and `Move Instance Method`, but also contains an algorithm for moving other members. The relations in size of the graphical components are not representative for their actual sizes. The refactoring we will describe is marked in red.

- Target type is not a static class: this complicates member accesses from the method.
- Target expression must be accessible at the call site of the method: otherwise prefixing the new method invocation with the expression will create a syntax error.
- Target expression is not null in the method or at the method invocation, and if the method contains a null-check of the target expression, the refactoring should be stopped. For an example of this, see example in Appendix 8.
- Target expression does not have side effects: inserting, and at runtime evaluating, an expression with side-effects will by definition change the behavior of the program.
- The selected expression or its aliases are not assigned to within the code that is reachable from the selection.
- The method cannot contain the `super` keyword: this can be unaccessible from another class.

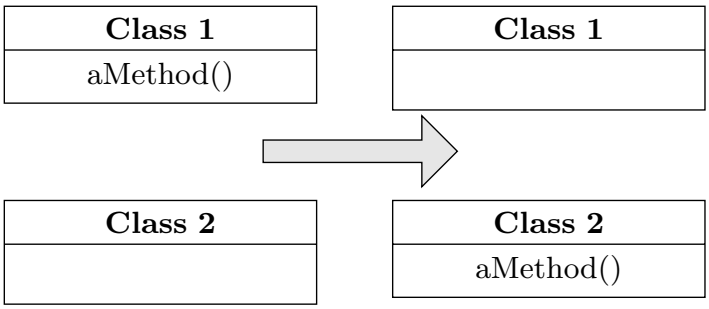

Figure 2.6: UML representation of `Move Method` used to describe the refactoring [10, p.142]. Left side shows a part of the system that should be refactored: two classes, one with a particular method and one without it. Right side shows the result of the refactoring: the method has been moved into the other class.

- Members referred to in the method must be visible from target type. Target expression is the exception to this rule.
- The program is well-formed, i.e. syntactically correct and type-checks (compiles).

The algorithm defined here is a modified version of Opdyke's `Moving Members into a Component (Aggregate Passed as Function Argument)`. He defines the refactoring for members, and we are interested in the function-part of it. Even though he defined it for C++ it can be generalized to Java, and we contrast the outcome of his refactoring with our algorithm in Listing 8. As we see in the listing, the result to the left can be preferred due to less problems with visibility of target member and possibly reduced coupling: if c was not needed by m it would not have been passed as a parameter. However, it introduces the subtle problem with behavior change, while the right side version does preserve behavior due to all field accesses being the same as in the original code, but this comes at the expense of more complex navigation paths and increased coupling, and can introduce visibility problems if target is not accessible. Opdyke's version is similar to the implementation in the Netbeans IDE, does produce broken code when applied with a private target [17, p.19]. This interpretation of the refactoring results in more complex code and pose the additional precondition that the target must be visible from the moved method, but it does avoid this particular behavior change. Observe that the only difference in the refactorings is the step shown in line 11 in Algorithm 3: instead of replacing the reference to target expression with an (implicit) reference to `this`, it is in Opdyke's version prefixed with c like the other referred-to members. This substitution will be the topic of Chapter 4 where we describe our remedy for the possible behavior changes.

2.7 Extract And Move Method

`Extract And Move Method` is a refactoring for extracting a selection into a method, moving it, and replacing the selection with the method invoked on

Original code

```
class C{
    private X x = new X();
    void f() {
      h();
    }
    public void h(){
        x.n();
        m();
        x.n();
    }
    void m(){x = new X();}
}
class X{
    void n(){print(hashCode());}
}
```

After Our Move Method

```
class C{
    X x = new X();
    void f() {
      x.h(this);
    }
}
class X{
    public void h(C c){
        n();
        c.m();
        n();//changed semantic
    }
}
```

After Opdyke's Move Method

```
class C{
    private X x = new X();
    void f() {
      x.h(this);
    }
}
class X{
    public void h(C c){
        c.x.n();
        c.m();
        c.x.n();
    }
}
```

Listing 8: Different interpretations of *Move Method* where the method h is moved with target x (highlighted). The result of the `Move Method` we will use is shown to the left, and of Opdyke's versions to the right, with code changes highlighted. The left side can reduce coupling (if c was not needed by m it would not be passed as a parameter), while the right hand side has increased length of the navigation paths increased coupling. We hide the methods n and m in the result to save space.

```
   input  : e – target expression of type E
          : m – an instance method in a class C
   output: m moved to E, updated method call
 1 declare a method n in E with same signature and name as m;
 2 replace invocation of m with e.n, same arguments;
 3 V ← all explicit or implicit this statements in n;
 4 if V is nonempty then
 5 |   add parameter C c to n;
 6 |   in n's invocation, give this as an argument to c;
 7 |   for v in V do
 8 |   |   replace v with a reference to c
 9 |   end
10 end
11 replace all references to c.e or e in n with this;
12 prefix all references to static members of C in n with a reference to C;
13 remove m from C;
14 import C and other types to E if needed;
15 remove unused imports from C;
```

Algorithm 3: `Move Method` for Java. Declare a new method in target type, copy signature and body, adjust member accesses in the new body and the method invocation.

a target expression. It can be useful to reduce duplication and method size, especially where members of another class is invoked repeatedly. It is very similar in definition and usefulness to `Move Method`, but precomposing with `Extract Method` allows it to be invoked on a selection instead of a predefined method. Thus, the target variable (which implies the target type of the move) and the selection must be found in the same scope.

We define the algorithm as first performing `Extract Method` as defined in Section 2.5, then `Move Method` on the resulting method, as defined in Section 2.6. An example of this refactoring is found in Listing 9. A high-level description is provided in Algorithm 4.

Preconditions. The preconditions on this refactoring are the union of the preconditions on `Extract Method` and `Move Method`. These are defined in their respective sections, and we will not list them again here. We rely on the assumptions we made when defining those refactorings to validate this refactoring, as described in their respective sections. Note that we have the same problem with behavior here as was discussed in Section 2.6, due to `Move Method`. This resembles the problem with `Extract Local Variable`, which will be discussed along with a solution in Chapter 4.

1: Original code

```
class C{
    X x = new X();
    public void f() {
        x.n();
        m();
        x.n();
    }
}
class X(){
}
```

2: Extract Method

```
class C{
    X x = new X();
    public void f() {
        h();
    }
    public void h(){
        x.n();
        m();
        x.n();
    }
}
class X{
}
```

3: Move Method

```
class C{
    X x = new X();
    public void f() {
        x.h(this);
    }
}
class X{
    public void h(C c){
        n();
        c.m();
        n();
    }
}
```

Listing 9: Example of `Extract And Move Method` with highlighted arguments. First `Extract Method` is applied to the selection argument, to extract the method; then we apply `Move Method` to the new method and the target argument. We assume n and m similar to in Listing 8.

input : e – target expression of type E, found in a class C
: S – selection
: *context* – the innermost, non-type scope containing S, declared in a class C
output: C with S replaced by a method invocation on e with the same behavior

1 Extract Method(C, S, *context*) to produce a new method m in C;
2 Move Method(e, m);

Algorithm 4: `Extract and Move Method` for Java, using the algorithms provided in Algorithm 2 and Algorthm 3.

Chapter 3

The Eclipse Project

In this chapter we describe the Eclipse Platform and how Java source code is represented and changed in the APIs supplied by Eclipse and the JDT.

Eclipse plays a core role in this work. We develop a plug-in for Eclipse, in the Eclipse IDE – and we do a case study on the Eclipse JDK UI project. The implementation of our refactorings is supported by the Eclipse API supporting program processing and transformation, and their implementation of the Java refactorings. This necessitates a thorough introduction to the different Eclipse components, so in this chapter we will describe the Eclipse SDK, the Eclipse JDT, the refactoring machinery provided by the Eclipse LTK, and present the implementation of Extract Method, Move Method and Extract Local Variable.

3.1 The Eclipse SDK

Perhaps the software most associated with the Eclipse name is the Eclipse SDK (Software Development Kit). This is a popular development environment built with a plug-in-architecture. There are several Eclipse SDK packages available, fitted for many particular needs[1]. All the packages have a common core, the Eclipse Platform[2], a language-independent foundation, providing the skeleton for a full SDK. The Platform defines the infrastructure required to support the use of Eclipse as a complete free-standing software development environment. It contains a standard workbench UI (User Interface), a project model for managing resources, debug infrastructure, etc. Language specific support is found in plug-ins, or sets thereof, like the JDT (Java Development Tools). Other plug-ins support other parts of the development process, e.g. the PDE[3] (Plug-in Development Environment), Eclipse Git Team Provider[4] or AspectJ[5]. A full list of projects can be found at the Eclipse web-page.[6] A combination of these can be plugged into the Platform to produce the smallest possible package that serves the developers needs.

[1] https://eclipse.org/downloads/
[2] https://projects.eclipse.org/projects/eclipse.platform
[3] http://www.eclipse.org/pde/
[4] https://projects.eclipse.org/projects/technology.egit
[5] https://projects.eclipse.org/projects/tools.aspectj
[6] https://projects.eclipse.org/

The Eclipse Project[7] is the commonly used umbrella term, encompassing the SDKs, IDEs and sub-projects. It is an open source project of eclipse.org, with the sub-projects developed in Git repositories.[8] The complete development effort of the project is a collaboration by multiple developers. Contributors face quality requirements: code must be nicely documented, and fixes or new functionality must be submitted with tests. Consequently, the projects come equipped with large test suites.

The Eclipse SDK for developing Java tools requires the JDT: a set of plug-ins that together with the Platform constitutes a full-featured Java IDE. The JDT plug-ins provide APIs for use and extension of themselves. Eclipse plug-in development is supported by the PDE subproject. This is a collection of tools, often categorized into the PDE Build, PDE UI and PDE API. Together they provide a comprehensive toolset that can be used to extend the Eclipse software.

3.2 The JDT

The JDT is a set of plug-ins providing the Java language specific development support. The plug-ins are categorized into:

- JDT API, providing tools for annotations and annotation-based build artifacts
- JDT Core, defining non-UI infrastructure like an incremental Java builder, a Java Model with an API for navigating the AST.
- JDT Debug, debugging tools for Java
- JDT Text, providing the Java editor with features like highlighting, code formatting, outline and code selection.
- JDT UI, implementing Java-specific workbench contributions, like the Package Explorer, Java Outline View and Wizards. The UI also provides refactoring support like Extract Method and Rename, with comprehensive wizards and previews.

In addition to this, the JDT has its own Java compiler, written entirely in Java.

The JDT contains tools for parsing and manipulating Java programs. These processes can be initiated through the provided API. The API offer a number of features, but we will focus on the ones that are relevant in this work.

The API is partitioned into several packages.[9] Here we list the ones relevant to us:

- `org.eclipse.jdt.core` contains The Java model, a set of classes that model the Java elements associated with a program. Examples of its types are ICompilationUnit, IJavaProject and IMethod.
- `org.eclipse.jdt.core.dom`, also referred to as the Java DOM/AST, is the set of classes that model the source code of a Java program as a structured document. This is a different program representation than the one

[7] http://www.eclipse.org/eclipse/

[8] https://git.eclipse.org/c/

[9] http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html

found in .core. Examples of types found here is AST, CompilationUnit, IMethodBinding, ASTParser, ASTVisitor, Expression.[10]

- `org.eclipse.jdt.core.dom.rewrite`, the set of classes used to make changes to an existing DOM/AST tree.
- `org.eclipse.jdt.core.refactoring`, providing an API for Java-specific refactoring framework facilities.
- `org.eclipse.jdt.core.util`, a set of tools and utilities for manipulating .class files and model elements.

The program representations provided by the `core` and `core.dom` are conceptually and practically different: `core` provides the Eclipse Java Model: an overview of organizational program elements, while `core.dom` provides an AST representation of the code, and both will be described in the next two subsections, with an example of a practical use in Section 3.2.3. The types found in both packages are inspired by Software Language Engineering (SLE) and Compiler Construction, and we refer to Chapter 2 for an explanation of the terminology.

3.2.1 The Java model

This section describes the Eclipse Java model defined in the `org.eclipse.jdt.core` plug-in.

The Eclipse Java model is a lightweight high-level representation of a Java project. It contains similarities to the AST, but does not contain the same amount of information, and only models the higher-level organizational details. It can be used to quickly provide an overview of a project, and then – through the use of `.utils` – allow lookup of specific model elements in the AST. Being light-weight makes it faster to generate, useful in e.g. the Package Explorer and Outline view, enabling quicker update. The Java model represents a Java program as a tree structure where each node can have one or more children. Figure 3.1 shows the structure of the model: the nodes on each level on the right side shows the allowed types that nodes on that level can have; left hand side shows the corresponding Java project elements.

These elements can be considered as abstract representations, also called *handles*, of the modeled elements. This means that the underlaying Java element does not necessarily exist even though we have its handle. The model only represents organizational details such as declarations of types, methods, fields and local variables. Other executable expressions like operators, assignments, keywords, field lookups and method invocations, has no representation here. This makes it useful for searching through and passing around the light-weight handles of the Java elements rather than their full AST. This model allows only very limited modification of the source code.

[10] Note that the type CompilationUnit found here, does not implement the ICompilationUnit found in the .core package.

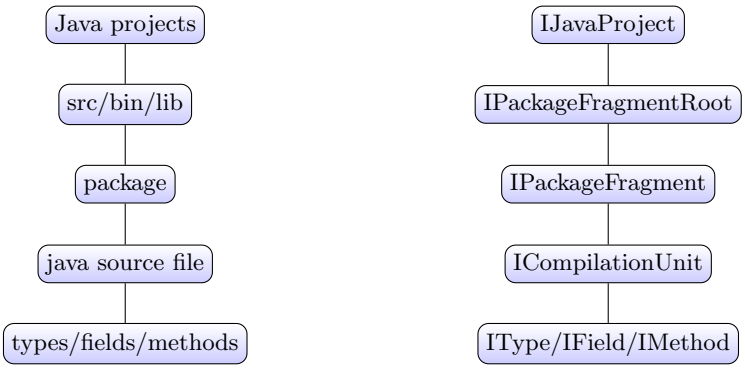

Figure 3.1: Project elements on the left, Java Model element to the right. Each node on the right represents the types that nodes on that level can have in the Java model, with the corresponding project elements to the left.

3.2.2 The Eclipse Java AST

This section describes the AST model of a Java program, as defined in the set of classes called Java DOM/AST, found in the `.core.com` package.[11]

The Java AST represented in this package corresponds mostly to any Java AST. It represents the Java source code as a structured document, where each program element is represented by an instance of thte class ASTNode. The nodes contains information like type bindings, variable bindings, its parent node (corresponding to the parent element), and the element's position and length in the source code.

The AST can be manipulated through the API found in the Rewrite-package, using an instance of the class ASTRewrite. It provides the infrastructure for code modification: the user defines changes of the AST, and the ASTRewrite object performs them and update the source code.

The AST type hierarchy closely resembles the one found in the Java model, but is more nuanced. It contains handles for all kinds of expression found in a Java program, including all types of members, expressions, operators and statements. A simplified type structure of a Java program AST is found in Figure 3.2. Here we use the notation from Section 2.1 and we exclude types like EnumDeclaration and AnnotationTypeDeclaration, focusing on the main part of regular Java programs.

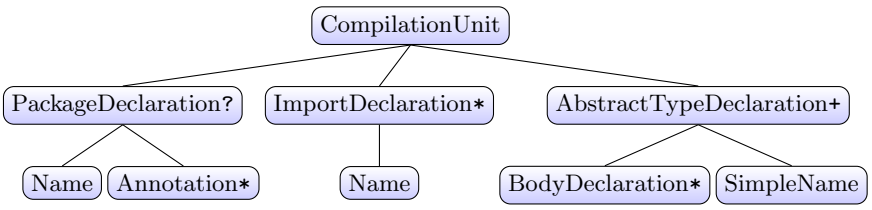

Figure 3.2: Simplified Eclipse AST representation of a Java Project

There will be one CompilationUnit that acts as the root node, but it can

[11] org.eclipse.jdt.core.dom

have any number of PackageDeclarations, ImportDeclarations and (one or more) AbstractTypeDeclarations, contained in separate lists. A PackageDeclaration always has a Name and an Annotation. An ImportDeclaration has a Name. An AbstractTypeDeclaration is a subtype of the BodyDeclaration: calling the method `getBodyDeclaration()` on a CompilationUnit will yield a list of AbstractTypeDeclaration instances, where each element can be cast to whichever of the three types its runtime type corresponds to. The TypeDeclaration can represent either an interface or a class. These does not have their own types, and are distinguished by the return value of the boolean method `isInterface()`. In either case, they can have JavaDoc, a Name, TypeParameters, a BodyDeclaration and a list of Fields and MethodDeclarations.

For a further example of the type structure from the level of MethodDeclarations and below, let us consider the example program `C.java`. Figure 3.3 contains a simplified version of the Eclipse AST of the method `C.f()`. As noted in Section 3.2 the types corresponds closely to the grammatical definition of a Java method in the Java programming language.

```
1 public class C {
2         public X x = new X();
3         public void f(X x) {
4                 x.m(this);
5         }
6 }
```

Listing 10: C.java

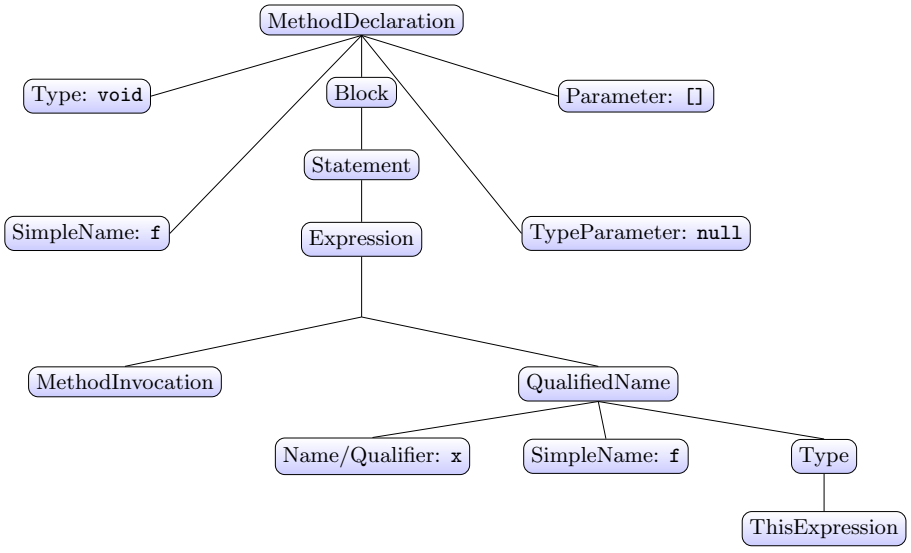

Figure 3.3: Simplified Eclipse AST representation of the method in Listing 10. We have excluded most of the empty members and annotations. The Block-node contains a list of Statements, and would have several children if the method body contained more statements than one. Parameter and TypeParameter also contains lists, currently empty ones.

To search or traverse the AST we can use the ASTVisitor class. The AST implements the Visitor pattern [11], with ASTVisitor offering the Visitor methods shown in Listing 11 for all the different ASTNode subclasses. Clients who want to traverse the AST can subclass ASTVisitor and override the appropriate methods. In Section 5.5 we describe how we use this approach to find the insert locations of our dynamic checks.

```
1 public abstract class ASTVisitor{
2     public void preVisit(Assignment node){
3     }
4     public boolean visit(Assignment node){
5         return true;
6     }
7     public void postVisit(Assignment node){
8     }
9 }
```

Listing 11: Parsing the source code into an AST

3.2.3 Parsing a Java code file in Eclipse

There are at any point two representations of our program available to us: the source code text version, and the AST version (the Java model can be considered a light-weight version of the AST, so they are not distinguished here). We are allowed to change either one, but we must go about it in different ways.

Through the API we can get access to the text document (IDocument) containing the code by specifying the path and file it should be loaded from, using the ITextFileBuffer. This gives us a plain text-representation of the program. This version does not contain any structure, and we can access elements by line number and character offset only, specifying which line in the document and how many characters into the line we want to look, as in a coordinate system.

Having the reference to the IDocument, we can instantiate a ASTParser to parse the text file into a AST. This will return a reference to the root, a node of type CompilationUnit. This is illustrated in Listing 12.

```
1 ASTParser parser = ASTParser.newParser(AST.JLS8);
2 parser.setKind(ASTParser.K_COMPILATION_UNIT);
3 parser.setSource(document.get().toCharArray());
4 parser.setResolveBindings(true);
5 parser.setProject(PROJECT_NAME));
6 parser.setUnitName(UNIT_NAME);
7 CompilationUnit cu = (CompilationUnit) parser.createAST(PROGRESS_MONITOR);
```

Listing 12: Parsing the source code into an AST

The subsequent AST can then be traversed using a subclass of ASTVisitor and edited using the ASTRewrite. If the node we want to edit is a list-node, for example as the Block node would be in Figure 3.3 if the method body had several statements, we use a ListRewrite, that is obtained from the ASTRewrite. If we want to insert statements (e.g. assert statements) we must generate it

from the statement subtree *from* the root node it will belong to (usually the ASTNode the ASTRewrite operates on) and specify the insert location to the ListRewrite by giving it a reference to the statement it should be put in front of:
`statementsListRewrite.insertBefore(assertStatement, statement, null);`
If the statement is not an element of the list, the method throws an exception. The user can create changes to the AST and add them to the ASTRewrite, but the AST is not changed until a change-method on the ASTRewrite is invoked. At this point the abstract changes described is turned into textual changes and the source code is updated – or an exception is thrown if the changes is impossible to perform. In section 5.4 we explain how this is used to insert the dynamic checks.

3.3 Eclipse's implementation of refactorings

In this section we give a brief descriptions of Eclipse's organization of the refactoring-related tools and a description of the relevant refactorings, as well as some detail of their implementation and intended use. We illustrate our descriptions with code examples when appropriate.

3.3.1 Language-Independent Infrastructure

The language-independent parts of Eclipse's refactorings are found in the Refactoring Language Toolkit (LTK), part of the `core` and `UI` parts of the Eclipse Platform.[12] This functionality is extended by code in the JDT to provide the Java-specific refactorings.

The infrastructure is mostly implemented in abstract classes, using the Template design pattern [11]: the subclass defines the specific functionality by implementing the abstract methods, but the abstract class ensures that the overarching algorithm is followed.

When implementing a new refactoring the following classes and methods should usually be subclassed:

- `RefactoringClass`: representing the specific refactoring, with three important methods:

  `checkInitialConditions(IProgressMonitor)`: invoked when launching the refactoring. Checks among other things the existence of a compilation unit and that its Java model structure can be determined.

  `checkFinalConditions(IProgressMonitor)`: invoked after `checkInitialConditions`, once all the arguments to the refactoring has been provided and checks the more complex preconditions.

  `createChange(IProgressMonitor)`: invoked after all preconditions have been checked and no fatal errors have been found.

- `RefactoringWizard`: presents the refactoring in the UI. The wizard shows the preview and error pages. It must implement the method `addUserInputPages`, describing the configuration page shown to the user.

- `RefactoringAction`: Launches the refactoring from the UI.

[12] `org.eclipse.ltk.core.refactoring`,`org.eclipse.ltk.ui.refactoring`

There are more classes that can be used, like the `RefactoringDescriptor`. This class uniquely describes a refactoring with both a human-readable description and an unique refactoring id.

LTK also supports *processor-based* refactorings, with one refactoring processor and any number of participants. Such a refactoring should extend the class `RefactoringProcessor` instead of `RefactoringClass`, but is otherwise similar to the refactoring described here. This is the case for the `Move Method` refactoring, its implementation described in Section 3.3.3.

The general lifecycle of a refactoring in Eclipse is as follows:

1. The refactoring is launched by a user or a script. An initial check is performed to determine whether the refactoring is applicable at all in the context desired by the user (`checkInitialConditions()`).

2. Configuration details is supplied by the user or script if necessary.

3. After all necessary information has been provided, an in-depth check is invoked (`checkFinalConditions()`) and the individual changes in the source text are calculated (`createChange()`).

4. The preview dialogue displays the changes; the user confirms and the LTK applies them to the workspace.

A last step can include adding an undo change to the IDE's undo history.

In the next subsections we will see how the JDT utilizes this framework in the implementation of the relevant refactorings.

3.3.2 Extract Method implementation

To invoke `Extract Method` from the UI, the user selects some statements, right clicks on the selection, and chooses `Extract Method` from the refactoring menu.

A general description of the functionality and usefulness of the refactoring `Extract Method` is given in the class `ExtractMethodRefactoring`.[13] It is instantiated by `ExtractMethodAction`, passed as an argument to the constructor of `ExtractMethodWizard`, which is responsible for checking the preconditions at the appropriate times: the initial preconditions are checked before the wizard loads, and the final are checked after the refactoring is fully configured.

To instantiate the refactoring the following constructor is called: `ExtractMethodRefactoring(ICompilationUnit cu, int selectionStart, int selectionLength)`. The arguments required are a handle of the compilation unit; the character offset where the selection argument starts, and the length of the selection. After initialization the name and visibility qualifier of the resulting method can be provided: the default name is "extracted" and the default visibility is private. The refactoring offer other configuration options, like whether method comments should be generated (false by default), which class the method should be extracted to in case of nested types (innermost one by default) and whether all additional occurrences of statements should be replaced (true by default).

The initial preconditions checks that the compilation unit exists and does not currently have any compile errors; that the selection exists, is a sequence

[13] `org.eclipse.jdt.internal.corext.refactoring.code.ExtractMethodRefactoring`

of statements, is not part of a control structure, and have at least one possible target to be extracted to. It also collects info about the parameters that will be needed, the referred variables, etc.

The final preconditions checks that the refactoring can be executed with the provided arguments, but does not check the initial preconditions again.

Finally, if there has been no registered `RefactoringStatus.FATAL` occurrences, the refactoring is performed and the change is added to the undo manager.

3.3.3 Move Method implementation

To invoke `Move Method` from the UI, the user selects a method, right clicks it, and chooses `Move Method` from the refactoring menu.

As described in Section 2.6 the `Move` refactoring is a collection of refactorings that encompasses several different ones. In Eclipse the subtypes are Move Instance Method, Move Static Member and Textual Move. This structure lends itself well to the processor based refactoring approach, and the overarching type is defined in the class `MoveProcessor`,[14] with `Move Instance Method` specified in the subclass `MoveInstanceMethodProcessor`. The functionality and usefulness of `Move Instance Method` is described in Section 2.6.

The refactoring is instanced in a helper class `RefactoringExecutionStarter`. Instantiating is done by calling the constructor
`MoveInstanceMethodProcessor(IMethod method, CodeGenerationSettings settings)`. The first argument is the handle of the method we want to move; the second is settings provided by the IDE, and should be `null` when invoked by scripting.

The instance of `MoveInstanceMethodProcessor` is passed as an argument to the constructor of the `MoveProcessor`, and both are passed on to the constructor of the `MoveInstanceMethodWizard`.[15]. The wizard is responsible for performing the required checks. The initial preconditions are checked before the wizard loads, and consists of checking that the method does not refer the superclass of its current type; that it does not refer enclosing instances, and doesn't contain any recursion. A failing check produces a `RefactoringStatus.FATAL`, and the user sees an explaining error message.

If there are no failed checks, the wizard loads, collects the the configuration arguments, and checks the final preconditions. These include checking that the target is found, is not a generic type, not read-only, and so on. If all checks turn out fine, the refactoring is finally performed and added to the undo manager.

3.3.4 Extract Local Variable implementation

To invoke `Extract Local Variable` from the UI, the user selects an expression, right clicks it, and chooses `Extract Local Variable` from the refactoring menu. The functionality and usefulness of `Extract Local Variable` is described in Section 2.4.

[14] `org.eclipse.ltk.core.refactoring.participants.MoveProcessor`

[15] `org.eclipse.jdt.internal.corext.refcatoring.RefactoringExecutionStarter .startMoveMethodRefactoring(IMethod method, Shell shell)`

`Extract Local Variable` is referred to as `Extract Temp` in the JDT implementation. It is defined in `ExtractTempRefactoring` and instantiated by the `ExtractTempAction` class, using the constructor `ExtractTempRefactoring (ICompilationUnit unit,int selectionStart, int selectionLength)`. A handle of the compilation unit the refactoring should be performed in is needed, as well as start position and length of the selection, similarly to `Extract Method`. Another constructor exists where the AST model of the compilation unit can be given instead. The instance is then passed as parameter to the `ExtractTempWizard` constructor, and the initial preconditions are checked. The initial preconditions here is that the selection is an extractable expression: that it is an expression of a non-void type and not part of a control structure. A failing check results in an error message, while success leads the wizard view to load. The configuration arguments are provided by the user: a variable name (the default name is deduced from the expression), whether all occurrences should be replaced (true by default), and declare the local variable as `final` (false by default). The wizard initiates the final check (that the name is not already used, and that the resulting compilation unit compiles) and performs the refactoring as well as adding it to the undo manager.

In Section 5.6 we describe how we take this refactoring as a starting point for our dynamic check generation.

Chapter 4

Proposed solution

In Section 2.3 we defined refactorings, discussed what it means that they should preserve behavior and how automated refactoring tools should ensure this property. In Section 2.4-2.7 we defined some refactorings with preconditions that are hard to check statically: `Extract Local Variable`, `Move Method` and `Extract And Move Method`, and looked at the result of applying them in cases where the preconditions fail. In Section 1.2.1 we briefly introduced our proposed solution to the precondition checking. In this chapter we will revisit the example of behavior change, look at some more examples, discuss the nature of the problem, and explain our solution and how it can be integrated in tools.

The problem with refactoring tools

As discussed in Section 2.3 refactoring tools should implement refactorings correctly – according to the general consensus of how the particular refactoring should behave – and preconditions must be checked thoroughly. Violations of the preconditions must be communicated to the user and the refactoring must either be aborted, or carried through without changing behavior. Some preconditions are impossible to ensure by static analysis in Java, and tool developers are left with the following choices:

- make overly restrictive tools that prevents the user from executing refactorings that might be correct.
- make tools inadequately restrictive tools that allows the user to execute refactorings that might be incorrect.
- to find other, non-static solutions to precondition checking and thereby ensure correctness.

While each of the first two options are discussed in Section 2.3.4, we will discuss the third option here. We will look at one such precondition, the problem it can cause and how we can design equivalent dynamic checks and utilize them in tools.

4.1 Extract Local Variable

In this section we discuss one particular behavior change that can happen when `Extract Local Variable` and `Move Method` or `Extract And Move Method` is performed, resulting from a missing precondition check. We find and formulate a property that must be true about the program for the refactoring to be behavior preserving, and move on to suggest how this can be checked dynamically in Java and how to integrate it into a refactoring. We start by investigating this behavior and precondition for `Extract Local Variable`, then generalize the results to the other refactorings.

Our working example in this section will be the code in Listing 13, which were also used in Section 1.2.1. We will provide more examples when needed. In the following we will use the names e of the extracted expression (here x), *temp* for the new local variable e is extracted to, and S to mean the selection in which e is replaced.

Original code

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      x.n();
6      m();
7      x.n();
8    }
9    void m(){x = new X();}
10 }
```

After Extract Local

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      m();
8      temp.n(); //semantic change
9    }
10   void m(){x = new X();}
11 }
```

Listing 13: Example of a behavior changing application of `Extract Local Variable`. The extracted expression is highlighted in the original code and replaced throughout the whole scope. Due to an assignment hidden in a method invocation the refactoring is behavior changing, but is still performed in Eclipse.

4.1.1 The precondition

We revisit the simple example used to show behavior violation in the case of `Extract Local Variable`, shown in Listing 13. The behavior stems from a violation of the precondition concerning assignment of the variable. We will refer to this precondition as P1, and recall the definition from Section 2.4:

"The selected expression or its aliases are not assigned to within the code that is reachable from the refactored selection."

This is a hard property to check statically, as static analysis in Java cannot confidently be used to determine which methods will be invoked at runtime, as discussed in 2.2.1. Another, easier to detect, violation of this precondition is shown in Listing 14 where e is assigned to directly in S. Typically IDEs can easily detect the presence of a such a *direct assignment* in the extracted

variable's scope, and Eclipse does not execute this refactoring. If we go ahead and do the refactoring manually we see that the assignment, if the lefthand side is replaced as well, produces a different behavior change than the one shown in Listing 13, namely that e is not assigned to any more and will retain its old value after the method finished executing. Alternatively, if we avoid replacing the reference on the left hand side of the assignment but otherwise perform the refatoring as before, then the resulting code of Listing 13 and 14 share the same behavior alteration.

Original code

```
1 class C {
2   public X x = new X();
3
4   public void f() {
5     x.n();
6     x = new X();
7     x.n();
8   }
9 }
```

Erronous Extract Local

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      temp = new X();//semantic change
8      temp.n();
9    }
10 }
```

Listing 14: Example of a behavior changing application of `Extract Local Variable`. This refactoring would be stopped in Eclipse, due to the direct assignment in the method body.

We see that neither replacing nor *not* replacing the occurrence at the lefthand side of the assignment yields a satisfactory result, and in the case of a direct assignment to e in the refactored scope the only behavior preserving solution is indeed to stop the refactoring and reporting the reason to the user. Thus we do not consider this problem further – although it would also be detected by our precondition checks – and turn our attention to the problem of discovering an assignment somewhere else that will be reached upon execution. In our simple example the field is assigned to by a method in the same class; this can of course be determined statically, but it cannot necessarily be determined whether this method is called from the scope we're refactoring. We will keep using this simplified example to save space in our Listings. More elaborate code structures can exist; if the field is non-private or can be assigned to through a method call on the object (like a set-method), the assignment to it can happen in several ways.

- The object of which it is a member of can be passed to another class – possibly in an external library, or in some code that is unavailable for analysis by the programmer and static tools – which can assign to it using the method or a direct assignment.
- A subclass can assign to the field it inherits
- Static access to a non-private field can allow any code with access to its declaring class to assign to it
- Use of the reflection API (introduced in Section 2.2) can change access and binding at runtime.

Thus, we conclude that **this precondition check must be performed dynamically at runtime**. It cannot be assured through static analysis in Java.

Original code

```
1  class C {
2    X x = new X();
3    X oldX;
4
5    public void f() {
6      x.n();
7      setOldX(x); // oldX = x;
8      m();        // x = new X();
9      resetX();   // x = oldX;
10     x.n();
11   }
12 }
```

After Extract Local

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      setOldX(temp); // oldX = temp;
8      m();           // x = new X();
9      resetX();      // x = oldX;
10     temp.n();
11   }
12 }
```

Listing 15: An example where the behavior before and after refactoring is the same, regardless of the assignment to the field x. During Line 8-10 *temp* and x points to different values, but *temp* is not evaluated during this time, and thus no harm is done.

We move on to consider how to formulate P1 as a runtime check. What we have discussed so far resembles the discussion of the *points-to analysis* [26, 28] : for all statements in the selected range, does the target expression e always evaluate to the same exact object? However, from the code example in Listing 15 we observe that the problem seems to be not that the expression is assigned to, i.e. that the set of its referred-to values has more than one member, but rather that one alias was assigned to and not the other, and consequently *the alias temp evaluating to a different value than e would do in its place.* If e is assigned some temporarily value, and then reassigned to its old value before *temp* is evaluated, it does not pose a problem, as the example in Listing 15 shows. Thus, we claim that P1 captures two types of behavior change: the one shown in Listing 13 and the one in Listing 14. As we consider only the first variant here, we formulate the following property that would ensure this behavior change to not happen:

$$e \textbf{ evaluates to the same value at all its occurrences in } S \tag{4.1}$$

Here, "all occurrences of *temp* in S" are exactly all occurrences of e replaced by *temp*, and we know, due to the statically available precondition check, that none of these are the lefthand side of an assignment. If they were, it would not pose a big problem, but it would affect the location of the dynamic checks, as we will see later. Based on this observation, and the fact that the introduced variable will have a fresh name, we reformulate the property in (4.1) as:

$$temp \textbf{ evaluates to the same value as } e \textbf{ at all occurrences of } temp \textbf{ in } S \tag{4.2}$$

If this holds, we can safely replace all occurrences of e with a reference to *temp* and trust that the result will be behaviorally equal to the original code.

Side effects: Note that we assume that evaluating e has no side effects. Examples of expressions with side effects are method calls that modify global or argument values, prints to screen, raises exceptions, and so on. We do not consider a get-method, nor field lookup, as something that can have side effects. Extracting an expression with side effects to a variable will usually alter the number of times the expression is evaluated: before, as many times as the expression occurs; after, only once when it is assigned to the variable. If an expression with side effects should be extracted and the programmer wish to preserve the number of invocations, it should be extracted to a method, not a variable, as in Fowler's discussion of `Introduce Explaining Variable` [10, p.124]. Note that this does not exclude our examples so far: the side effect is of the method that is called *on* the extracted expression, not a side effect of evaluating said expression. In the rest of this chapter we will assume the extracted expression to be free of side effects.

4.1.2 Expressing the property

In this subsection we explain how we can state the condition formulated in (4.2) in Java. We continue using the names e for the extracted expression and *temp* for the local variable.

How we state that *temp* and e evaluates to the same value depends on how we define the "same value". For some types, this can be checked by the `equals`-method, but in some cases that might not be enough – and it would require the extracted types to implement an appropriate equals method – and we instead express it using the stronger property: the two expressions are *reference equal*, or, using Java syntax: `temp == e`.

This is the assumption under which we substitute every reference to e with *temp*, and it is a property that must hold for the program to be correct afterwards. Such a property is exactly what is the suggested use of the `assert` keyword (introduced in Section 2.2), and we suggest the following expression in Java:

```
assert temp == e;
```
(4.3)

ideally with an explaining error message detailing the error and the refactoring that caused it: "Extract Local Variable caused a behavior change due to reassignment". This will communicate the problem to the programmer, and allow her to decide for herself if the refactoring should be undone or if this is an acceptable change in semantics. Using assertions for this allows the programmer to disable and enable them as she sees fit, and when they are disabled the arguments will not be evaluated and does not affect the execution of the program. We consider it and advantage that the checks intrude as little as possible in the program execution and is clearly distinguishable from the program itself.

To test the asserts the introduced statements must be executed. This requires the program to be run, for example by a test suite, but relaxes the requirements of the tests. Usually, as discussed in Section 2.3 the test suite is responsible for discovering the errors a refactoring has introduced by testing the program against its specifications. Introducing our dynamic checks requires a test suite to cover the code, not to check the outcome of running it. As such, these checks can be seen as an addition to a test suite, and additional documentation of the program. Running the code can also be done by normal execution

without using tests. Finally, after the checks have been run, the programmer can remove all asserts if she wishes, either manually or automatically through the refactoring tool. It would make sense to supply an option in the tool for doing this automatically, as well as the option to undo the whole refactoring.

4.1.3 Location of assert insertion

We have established that the property in (4.3) must hold every time we evaluate *temp*. This leaves us with two approaches to inserting the checks: as the last statement executed before the reference is looked up, and as a wrapper of the reference lookup itself. Both has benefits and drawbacks which we will explore.

Statement

Using the code example from Listing 13, we illustrate in Listing 16 the result with asserts added. We insert the assert statements in the refactored code as *the last statement to be evaluated before each statement that contains the temp reference*. This is equivalent of inserting every time a substitution happens. Alternative locations will be discussed later. In this example the asserts captures the behavior change while communicating clearly to the programmer what is wrong. The instrumentation of the code is done as non-intrusive as possible, by only adding statements to the method body at appropriate locations with no other modification to the program. If evaluation of asserts are disabled this code is semantically equivalent to the refactored code without dynamic checks. The algorithm for inserting the asserts is described in Algorithm 8 as part of the algorithm for `Extract Local Variable`.

This approach produces checks that are sound, but not complete. If the check fails there are indeed a behavior change (even though the programmer might decide that it is an acceptable, or even intended, change), but if it succeeds we cannot be sure that the condition holds. Several methods can be invoked during the evaluation of a single statement: e can be assigned to, evaluated, and reassigned to the original value all during the evaluation of one statement, as shown in Listing 17. As such, the algorithm in Algorithm 8 provides a heuristic and serves as a conceptual algorithm, but there are still particular code structures for which it will not detect the violated condition. Alternatively the code can be instrumented more intrusively with method wrappers.

input : e: an expression of non-void type E
input : S: a selection, as a list of consecutive statements
input : *context*: the outermost, non-type scope containing S
output: *context* with e safely extracted to a local variable in S
1 $v \leftarrow$ fresh variable name;
2 *assertStatement* $\leftarrow$ assert e == v;
3 $V \leftarrow$ all occurrences of e in S;
4 **for** $e2 \in V$ **do**
5 | insert *assertStatement* just before the statement containing $e2$;
6 | replace the occurrences of $e2$ with v;
7 **end**
8 add a new variable declaration `E v = e;` to *context* just before S;

Algorithm 5: Modified Extract Local Variable to generate and insert assert statements

Refactored source

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      m();
8      assert x == temp: "Extract Local changed semantic";
9      temp.n(); //semantic change
10   }
11   void m(){x = new X();}
12 }
```

Listing 16: The result of applying `Extract Local Variable` to the working example: dynamic checks are added and will alert the programmer at runtime.

Wrapper

Another, more intrusive way to instrument the code with the dynamic checks, is to implement them in a method which is then wrapped around the reference to *temp*. Every time *temp* is evaluated in the code, as in the example of a method invocation `temp.m()` we wrap the reference with a call to a method that checks the condition from (4.3), throws an assert error if the check fails, and returns the value of *temp* otherwise: `wrapAssert(temp, x, errormessage).m()`. This allows us to check the value itself, not before or after, and provides a sound and complete – in this context – precondition check for the condition stated in (4.2).

This requires more changes to the checked program: a method must be declared in the refactored type, which should ideally be possible to reuse in the case of several refactorings checked at the same time (this can be relevant e.g. when doing a global check of which selections can be extracted to a new method, as described in 4.2). Thus the method should be of a generic type and take a string error message as argument, as well as the extracted expression e for comparing. An example is shown in Listing 18. Note that a significant difference between this solution and only inserting the statement, is that the method will be invoked and its arguments evaluated even if asserts are disabled. This may or may not be a problem, but should be considered. Introducing a new method also requires general precondition checks concerning whether the signature is already present [10].

Original code

```
class C {
  X x = new X();

  public void f() {
    if(x.n()
        && setNewX()
        && x.n()
        && setOldX)
        {
    }
  }
}
```

After Extract Local

```
class C {
  X x = new X();

  public void f() {
    X temp = x;
    assert temp == x;
    if(temp.n()
        && setNewX()
        && temp.n()
        && setOldX())
        {
        assert temp == x;
    }
    assert temp == x;
  }
}
```

Listing 17: Example of a behavior changing application of `Extract Local Variable` where the assignment, evaluation, and reassignment happens in one statement. We assume `setNewX()` assigns a new value to the field x and `setOldX()` reassigns the old value to x. The full example can be seen in Appendix A. Our assert statements, inserted all possible places in this code, cannot capture the change since it happens during a single statement. Here we have used an if-clause for the statement, but it could have been several other types of statements as well.

Assert Wrapper

```
public class C {
    X x = new X();
    void f(){
            X temp = x;
            if(wrapAssert(temp, x, "Extract Local").n()
                && setNewX()
                && wrapAssert(temp, x, "Extract Local").n()
                && setOldX()){
            }
    }

    private <T> T wrapAssert(T temp, T t, String error){
            assert temp == t : error;
            return t;
    }
}
```

Listing 18: The same example as in Listing 26, but with the checks encoded as generic method wrappers instead of assert statements. Generic, because the method can then be reused for other expressions of other types in the same compilation unit. The evaluation of the assert statement can still be turned off and on, but the method *wrapAssert* will nonetheless be invoked during all runs and its arguments evaluated.

4.2 Extract And Move Method

In this section we consider a similar behavior change to the one discussed in the previous section, but this time resulting from an erroneous application of `Extract And Move Method`, defined in Section 2.7. We generalize the solution described for `Extract Local Variable` to also work for this refactoring, and we suggest ways of implementing this in a tool. Our working example in this section will be the original code in Listing 13 (same as in Section 1.2.1 and the previous section), with `Extract And Move applied`: the full process was seen in Listing 9. We will investigate the code changes carefully. In the following we will use the names e of the extracted expression (here x) and S to mean the selection in which e is replaced.

4.2.1 Similarities to Extract Local Variable

We point out that `Extract And Move Method` is conceptually somewhat similar to `Extract Local Variable`, and have some of the same challenges. As mentioned in Section 2.6 `Move Method` can behave differently based on the selected method. For example:

- If the selected method is not invoked anywhere and does not refer any members of any class, it can be moved by copying and pasting the text, and updating import declarations.
- If the selected method refers members of class it currently resides in, C, but not its target class, the new method needs an extra argument of type C through which it can access the members.
- If the selected method refers members of the target class, E, it must be decided whether it should take an argument through which it will access the members (which increases the complexity of their qualifier) or if it should access the members directly using the `this` keyword.

In the behavior changing example in Listing 19, the last is the case. In previous examples the `this` keyword was not written explicitly when used as a qualifier. In Listing 19 we give an extensive example of `Extract and Move Method` where `Extract Local Variable` is applied first and the `this` keyword is written explicitly to highlight the substitutions that occur. Note that the same result is obtained by applying Opdyke's behavior preserving `Move Method` [23] (described in Section 2.6) and then substituting every occurrence of $c.e$ (here e is the field x) with `this`. The substitution step would then introduce the same behavior change as extracting to a variable does in Listing 19. In the algorithm given in Section 2.6 this is exactly how the refactoring is described. We conclude that *the behavior change is introduced by substituting references to e or $c.e$ in the method with* `this`.

1: Original code

```
class C{
    X x = new X();
    public void f() {
        this.x.n();
        this.m();
        this.x.n();
    }
}
class X(){
}
```

2: Extract Local

```
class C{
    X x = new X();
    public void f() {
        X temp = x;
        temp.n();
        this.m();
        temp.n();
    }
}
class X(){
}
```

3: Extract Method

```
class C{
    X x = new X();
    public void f() {
        X temp = x;
        this.h(temp);
    }
    public void h(X temp){
        temp.n();
        this.m();
        temp.n();
    }
}
class X{
}
```

4: Move Method

```
class C{
    X x = new X();
    public void f() {
        X temp = x;
        temp.h(this);
    }
}
class X{
    public void h(C c){
        this.n();
        c.m();
        this.n();
    }
}
```

Listing 19: A similar example to Listing 8 where `Extract Local` is performed first, and explicit use of `this` keywords. We assume n and m as before. The behavior change is introduced already in the first step, by `Extract Local Variable`. From step 2-4 the behavior is preserved. A finished step can be to inline *temp* in f.

4.2.2 Generalizing the property

If we consider `this` a name to be evaluated (instead of a keyword) it is rather similar to the substitution in `Extract Local Variable`: in the original code we evaluate the field every time it occurs, while in the refactored code we evaluate it once and later refer to that value by a local reference either *this* (essentially `this = c.x` or *temp*. Thus, we conclude that *the behavior change introduced by the substitution step in* ***`Extract And Move Method`*** *is the same as in the substitution step of****`Extract Local Variable`***. Based on this, it is straightforward to generalize the solution for `Extract Local Variable` to work for `Extract And Move Method`, and we formulate the following general property, P, for a behavior-preserving substitution:

When a reference is substituted for another, they must evaluate to the same value at every substitution.

This is the same property we check for `Extract Local Variable`, and consequently we can use `Extract Local Variable` as a precondition check for this property of `Extract And Move Method`:

If P holds for `Extract Local Variable` applied to e and S, then it will also hold for `Extract And Move Method` applied to the same arguments.

This result allows us to formulate the following property, P3 for an application of `Extract And Move Method` to a field e and well-formed selection S:

```
assert this == c.e;
```

If e is not a field, then it can only be assigned to using a direct assignment in S, which can be discovered by static analysis. The results described here applies to `Move Method` in general, but becomes more complex when the method is invoked several places. We will revisit this when we describe the integration of this precondition check.

4.2.3 Integration into a refactoring tool

Checking P3 for an application of `Extract And Move Method` can be done either as a freestanding precondition checking step or as a part of the refactoring.

Extract And Move With Asserts

When the user invokes `Extract And Move Method` on a target and selection argument, it can (after the static precondition checking) be applied with P3 inserted in the code either as assert statements (as shown in Listing 20) or as wrappers. Then the resulting code must be executed to reveal any introduced behavior changes, in which case the programmer decides if the refactoring should be undone. If no behavior change is discovered, or the programmer decides the change is acceptable or intended, the asserts are removed (ideally by the tool) and the refactoring is finished.

Ghost variable Notice that introduced checks needs access to the target field. If e is a local variable or a parameter it can only be assigned to by a direct assignment in the body it is declared in, in which case the refactoring should be stopped by the static precondition checking. Thus, we assume that we are only introducing the dynamic checks in the case of fields. These fields must be accessible from the target method, either directly or using get-methods. In an implementation an alternative could be to generate special get-methods for this specific use, and discard them afterwards. However, assuming e is a directly accessible, non-static field, we still need the instance the new method is invoked from to access it. Thus the introduction of a new parameter specifically for this purpose. In our implementation this is a regular parameter, but it can be thought of as a *ghost-* or *modeling variable* [12]: a variable introduced only for the purpose of checking a property, not for general use. This is future work. Here we simply introduce a new parameter to the method which should be invoked with the argument as *the qualifier of the target expression*, i.e. `this` (as shown in Listing 20). This variable should be discarded along with the asserts. If e is a static field it can still be accessed through the object, although it would be enough with the qualifying type.

Original code

```
class C {
  X x = new X();
  public void f() {
    x.n();
    m();
    x.n();
  }
  void m(){x = new X();}
}
class X{
}
```

Extract And Move with asserts

```
class C {
  X x = new X();
  public void f() {
    x.h(this, this);
  }
  void m(){x = new X();}
}
class X{
  public void h(C c, C c2){
    n();
    assert this == c.x;
    c2.m();
    assert this == c.x;
    n();
  }
}
```

Listing 20: Example of a behavior-changing application of `Extract And Move Method`. Extracted selection is highlighted with x as target; the method n is not shown. The behavior change will be picked up by the introduced asserts. Notice c can be thought of as a ghost variable and $c2$ as a regular parameter.

Extract Local Variable as precondition

An alternative to the solution we just described, and one that does not require access to the field from the target type, is to use the property P, that correctness of the substitution `Extract Local Variable` predicts correctness of the substitution in `Extract And Move Method`, and perform the precondition check using `Extract Local` as a proxy. In this case `Extract Local Variable` with asserts is executed on the target selection and expression, the result run, and if no behavior changes is discovered the asserts may be removed, the local variable inlined, and the intended `Move` refactoring applied as usual. The preconditions on the arguments to `Extract Local Variable` is more relaxed than to `Extract And Move Method`, so all application of the latter will also be a valid application of `Extract Local Variable`.

As mentioned this precondition can also be used for the general `Move Method`. In that case it is a precondition on all target arguments: if the method is invoked several places there can be many of them. It can also be used to determine possible refactoring targets in a class: this is where the generic type of the wrapper method comes in play: one wrapper method can be generated for each class (or a utility method to be used in all classes).

Chapter 5

Implementation Details

For our experiment (described in Chapter 6) we developed the Safer Refactoring plug-in for Eclipse. We base our work on a previous master thesis [17] for which the `Extract And Move Method` plug-in was developed.

Our main contribution to the plug-in is reimplementing the `Extract Local Variable` from the JDT, and modifying the already present Extract And Move Method, to generate the dynamic checks. To increase the quality of the refactored code we improved the `Extract And Move Method` heuristic and added a number of more complex test cases. To facilitate this development we did a large refactoring of the source code, making it easier to introduce our – and future – functionality, and improved documentation of the overall project.

In this chapter we introduce the plug-in that was available to us when we started. We then describe the resulting plug-in and how the dynamic checks are integrated in the refactoring. The assertions are introduced in different locations for the two refactorings, and we detail both. We briefly describe the major modifications done to the original plug-in to facilitate our development, and the extra test cases. A more detailed catalogue of special cases and examples can be found in Appendix A as well as in the the Git repository with our Eclipse-based Java refactorings at `git://git.uio.no/ifi-stolz-refaktor.git`.

5.1 The Extract And Move Method Plug-in

In this section we introduce the thesis and source code that was taken as a starting point of our implementation.

The `Extract And Move Method` Plug-in was developed by Kristiansen in his master thesis Automated Composition of Refactorings: Implementing and evaluating a search-based `Extract And Move Method` refactoring [17]. His aim was to develop an efficient, safe, quality-improving and helpful automated `Extract And Move Method` refactoring that could be applied in large scale on source code. Although not quite finished, the plug-in offers a composed version of the Eclipse refactorings that can be invoked on different levels in the code: either the user selects statements in the source code, right clicks and invokes the refactoring through the menu, and an analyzer decides the target of the move operation; or the user invokes it on a method and the analyzer finds the optimal selection and target from the method bod. The refactoring can be applied on any level

	Old Extract and `Move Method`	New Extract and `Move Method`
Executed refactorings	2469	755
Resulting compile errors	322	14
Tests failing before	3	0
Test errors before	4	0
Test errors after	565	84
Tests failing after	5	161

Table 5.1: Experiment results from the old and new `Extract And Move Method` refactoring. The new one includes assert generation, which excludes a large number of refactoring opportunities. An error indicates that the test did not run correctly (usually due to an error or exception), while a failure indicates a different outcome of the test than expected (possibly also due to error or exception, but of the wrong type). This and the rest of the results are covered in Chapter 6.

in the hierarchy from methods to projects, invoked from the Outline View or Package Explorer. Invoking it on a higher-level element like a package or project is equivalent to executing the search-based refactoring once on each method in the subtree. The machinery of the code transformations was in large borrowed from Eclipse's implementation, with custom infrastructure build between this modified and composed version of Eclipse's refactorings and the various points call sites of the Eclipse API and workbench.

The plug-in was considered incomplete at the end of the project. The author describes how executing the refactoring on the Eclipse JDT UI project produces a high number of compile errors, and about the plug-in he writes [17, p.93]:

"The pre-refactoring analysis of the `Extract And Move Method` refactoring is currently not complete enough to make the refactoring useful. It introduces too many errors in the code, and the code may change its behavior."

In his experiment section a similar case study to our is reported: the tool is applied to the Eclipse JDT UI source code, and afterwards its test suite is run. The results are summarized and contrasted with ours in Table 5.1, and an account of the origin of our errors, as well as an explanation of the decrease in performed refactorings, is found in Chapter 6. Note that we modified the heuristic both to produce less compile errors in general, but also to work specifically with our assert generation, which excludes a large class of targets. The New `Extract And Move Method` in Table 5.1 is performed with dynamic checks. In addition to the shortcomings described by the author, the plug-in supported only this one refactoring, and most inner functionality was not suited for reuse in the context of a new refactoring.

Our work is based on the Automated Refactoring thesis: the idea of inserting assert statements when refactoring to expose behavioral changes is from Kristiansen's Future Work, and we were able to reuse large parts of his tool directly or with modifications.

5.2 The Safer Refactoring Plug-in Overview

In this section, and the following four sections, we describe the plug-in we developed. This section contains an overview of its implementation partitioned into four parts, each described in-depth in its own section.

In regards to functionality the plug-in resembles its predecessor. The user can still invoke refactorings on different levels in the source code, and the plug-in will determine the missing arguments. It now offers the `Extract Local Variable` refactoring, which can be invoked on an expression, a method or a project. It still offers the Extract And Move Method, which can be invoked on a method or a project. Both refactorings add the dynamic checks described in Chapter 4 to the source code..

The plug-in can be partitioned into four parts:

- Analyzer: for analyzing the input code to produce refactoring arguments, described in Section 5.3
- Refactoring: the implementation of the refactorings, assert generation, described in Section 5.4-5.6.
- Infrastructure: exceptions, logging, benchmarking, with our modifications described later in this section.
- Plug-in specification: manifest file, event handlers and UI, briefly explained in the following.

The user interaction is described by the plug-in specification. This part defines and connects the available menus and views with their respective Java handlers. In the case of a menu item being invoked, the Eclipse- and Java context is passed to the appropriate, refactoring-specific, handler, i.e. a specialized Java class. We removed some parts of the UI, and introduced new menu items for the new refactoring options, but did not perform large changes to this part, and thus do mention it again. The infrastructure provides utilities for getting the right elements (like methods, packages, selections) from the context as well as the overall data flow and some unit tests and test cases, and is implemented using generic types and factory objects to facilitate reuse. Thus the current plug-in can be extended to support more refactoring types without changing the infrastructure implementation or data-flow skeleton at all. The appropriate Java elements are passed on to the appropriate analyzer, which applies a heuristic to find missing arguments. The analyzers reside in factory-classes, such that the Java element is passed to the factory which returns the refactoring arguments, and are refactoring-specific in that they must find and filter targets based on the refactoring. The refactoring part refers to the code that executes the refactorings, i.e. takes the arguments provided and the available program model and produces the change that can be applied to the AST through Eclipse's infrastructure. This includes generating dynamic checks. This part uses the Eclipse API to generate AST changes, and our infrastructure for error handling and logging.

Figure 5.6 contains a diagram of parts of the execution paths when either refactoring is invoked on a method. This diagram will be referred to throughout this section.

5.2.1 Infrastructure

Here we describe the relevant part of the infrastructure of the plug-in, and the changes we did to it.

The non-UI, non-refactoring, non-analyzer parts of the plug-in is what we call infrastructure. It is responsible for large parts of the data flow, and if a refactoring is invoked on a project, it is responsible for splitting it up into packages, compilation units, types, and finally methods, and then to execute the refactoring once on each method.

We made the architecture more modular, to avoid passing excessive data around. We introduced the factories that acts as a black box to encapsulate the analyzer, and allows us to use generic types throughout the whole inner structure of the plug-in. In doing this we could reuse large parts of the code (with modifications), and can now use the same skeleton for different refactorings. This will allow the plug-in to be easier extended in the future.

5.2.2 Development software

All development required for our experiment was done in the Eclipse Java IDE for Plug-in development. This IDE relies on the JDT for parsing and representing source code, which gave us access to an extensive API [1] where we could perform transformations on the AST of the UI code. We developed our plug-in for the Eclipse IDE, in the Eclipse IDE, using its Java model, AST and refactorings. For development we used the following SDK:

- Eclipse for RCP and RAP Developers

  Version: Mars Release (4.5.0)

  Build id: 20150621-1200

5.3 Analyzer

In this section we describe the part of the plug-in responsible for analyzing source code and finding refactoring arguments. When a refactoring is invoked on a program element like a method or project, the analyzer determines both the target and the selection. A generic `RefactorAnalyzer` class iterates over all possible selections to find the "best" one. The specific analyzer algorithm is invoked once on each selection to find the best target in that selection, which is then used to determine the optimal selection. The heuristics for finding the best target and the best selection are motivated by the software quality improvements mentioned in Section 2.3.

Code Structure. The algorithms used in these algorithms are very similar for the two refactorings, but not quite identical. The refactoring-specific implementations of the analyzes are split into separate classes, with common functionality in an abstract superclass and shared helper classes. The helper classes consist of different ASTVisitors for collecting Prefixes (candidates for the target expression) and Unfixes (expressions that are assigned to by a direct

[1] http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Foverview-summary.html

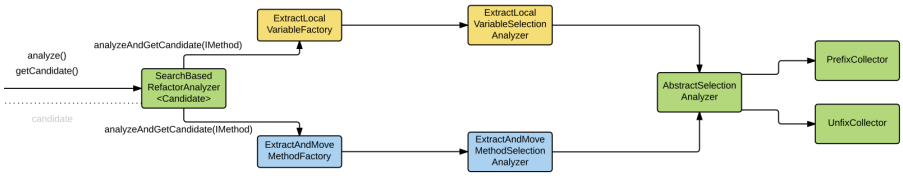

Figure 5.1: The green `RefactorAnalyzer` is part of the generic infrastructure, and invokes the analyzer through the factory objects. The blue classes contains functionality specific for `Extract And Move Method`, yellow for `Extract Local Variable`, and the green classes are used by both refactorings. The top candidate is returned by the factory, and returned to the RefactorExecutor, shown in Figure 5.6.

assignment in the refactored scope). These classes are the `PrefixCollector` and `UnfixCollector` shown in Figure 5.1, extended by specialized subclasses to provide specific functionality. The algorithm uses a heuristic for ranking target and selection candidates and excluding unsuited ones. The analyzer is invoked from the refactoring infrastructure through a factory object, and the internal data of the analyzer is not accessible from the rest of the code. This is shown in Figure 5.1, and is a rather big change in architecture from the original code in which the analyzer object was passed around. This architecture was a better fit for our larger code base, as it is more modular with clearer data flow, making it easier to maintain. However, if one would like to have access to all possible – or the top – candidates, the previous setup could be beneficial.

Naming. Our algorithm is not responsible for finding a meaningful name for the variable or method: the problem of generating names that convey an element's meaning to a programmer is hard, and one that relates to user-friendliness of the plug-in. Our main concern for now is the uniqueness of the generated names, to avoid the name-related preconditions of the refactorings [10], so we deem meaningful names outside the scope of this thesis, and generate our names as a large, pseudorandom number (generated by the `nextLong()` method of the Java `Random` class), prefixed with `generated_` or `temp_`. This method leads to names that are very likely unique, but not meaningful at all.

5.3.1 The Target Argument

The expression argument to the refactoring, or the target argument, is described in Section 2.5-2.7. For `Extract Local Variable` this is the expression we extract; for `Extract And Move Method` this is the expression we invoke the new method on. The general heuristic for finding the "best" target e from a selection S is as follows: Let T be the set of possible targets, defined as all subexpressions p of the elements in S, such that:

- p is a Prefix
- p is not an Unfix

- p is not *unsuitable* as a target for extraction, i.e. refactoring-specific conditions imposed on the arguments

What constitutes a Prefix and an *unsuitable* target differs for the refactorings, and will be described as we turn to each refactoring.

Extract Local Variable

We wish to include get-methods in the set of possible targets for Extract Local Variable, and so we extend Kristiansen's [17] notion of Prefix to include what we believe is a get-method, but only as the first segment of the prefix. We extended the `Extract Local`-specific part of the `PrefixCollector` class (implemented using the Visitor Pattern [11]) to include a method invocation iff:

- the name starts with "get", and
- the method declaration that the binding resolves to (statically) contains only one line, and
- that line is only a return statement.

This check was implemented using the Visitor Pattern [11] in the class `GetterWithoutSideEffectsVisitor`. When the precondition collector visits a method invocation we check the name, and if it starts with "get" we resolve the static binding of the method and traverse its AST with an instance of the getter-visitor. It checks the statically bound method for the two other conditions, and if they hold, we add it to the prefix set. We believe this resembles what a programmer would do.

For this refactoring, an *unsuitable* target consists of prefixes of segment length one that is referred only once. The expression is then already extracted. Assigned-to expressions are already excluded by being an Unfix.

Extract And Move Method

For this refactoring we use Kristiansen's [17] notion of Prefix, but for the *unsuited* predicate we add more cases. A Prefix p is *unsuited* if:

- p has only one segment *and* occurs only once.
- p is a local variable, it cannot be changed by a method call, so unless it is assigned to in the selection (thus an Unfix) this refactoring will not change the semantic of the program.
- p is a field, it has to be visible from the resulting method n, otherwise we cannot generate syntactically correct assertions. This can be remedied by generating getters, but we did not pursue this idea yet, and we use a heuristic to check that p is visible.
- p's type has generic type arguments then it can be hard or impossible to move the method.
- p is a static class that will make the moved method static, which can complicate member references in it.

The list of properties that makes a Prefix *unsuited* is a list we have compiled by iterating through the compilation errors produced by the plug-in, and adding filters that captures the different types of errors.

Ranking the targets

After the Unfixes and *unsuited* targets are removed from the set of prefixes, we rank the remaining expressions by number of occurrences in the selection and number of segments. The top ranked Prefix will be considered the best target candidate to the refactoring.

We define a total order on T where the targets are sorted lexicographically by how many times they are referenced in the section, then by segment length, and choose e from the maximal elements of T.

In the implementation, the target candidates are stored in a PrefixSet, so if there are several maximal elements it is arbitrary which one is picked. Note that this makes the refactoring non-deterministic, and can make the number of applied refactorings vary due to the added checks we pose refactoring: they may lead to the refactoring being stopped, in which case a new target is not chosen. This last option, as well as an *repeated, exhaustive* refactoring of each method is considered future work. If there are no candidates the refactoring of this method is aborted.

5.3.2 The Selection Argument

The heuristic described here is taken from Kristiansen's thesis [17]. We have extended it to also work with Extract Local Variable. The selection argument to the refactorings are described in Section 2.4-2.7. For `Extract Local Variable` this is the statements in which occurrences of the target will be replaced; for `Extract And Move Method` this is the body of the extracted method. If the user invokes `Extract Local Variable` directly on a selected target expression, then the whole method body will be taken as the selection argument. If a refactoring is invoked on a higher-level program element, the algorithm will find one selection argument per method, along with its target argument. The algorithm for the target is described in the previous section, and used by this heuristic.

Assuming as input a method m, we will find the selection argument S as follows:
Let C be the set of all selection candidates for S, containing all possible continuous selections in m's method body. We exclude selection candidates that are not well-formed, as described in Section 2.5-2.7. For each remaining candidate we use the collector classes [17] modified for our use and the algorithm described earlier to find the optimal target expression, the number of Unfixes and target candidates. We then rank the selection candidates, c, lexicographically by the following conditions, listed in order of decreasing importance:

- c contains no Unfixes.
- c contains only one possible target.
- c's target expression has a high number of occurrences and segment count (same as the ranking of the targets).

In the implementation, the order of the candidates are based on their position in the code so if there are more possible candidates, the one that occurs first in the code will be picked. If no candidate is found the refactoring is aborted.

5.4 Refactoring

This section describes the part of the plug-in that implement the two supported refactorings (defined in Chapter 2) and the dynamic check integration into these implementations, as described in Chapter 4. The relevant classes and the execution path is shown in Figure 5.2.

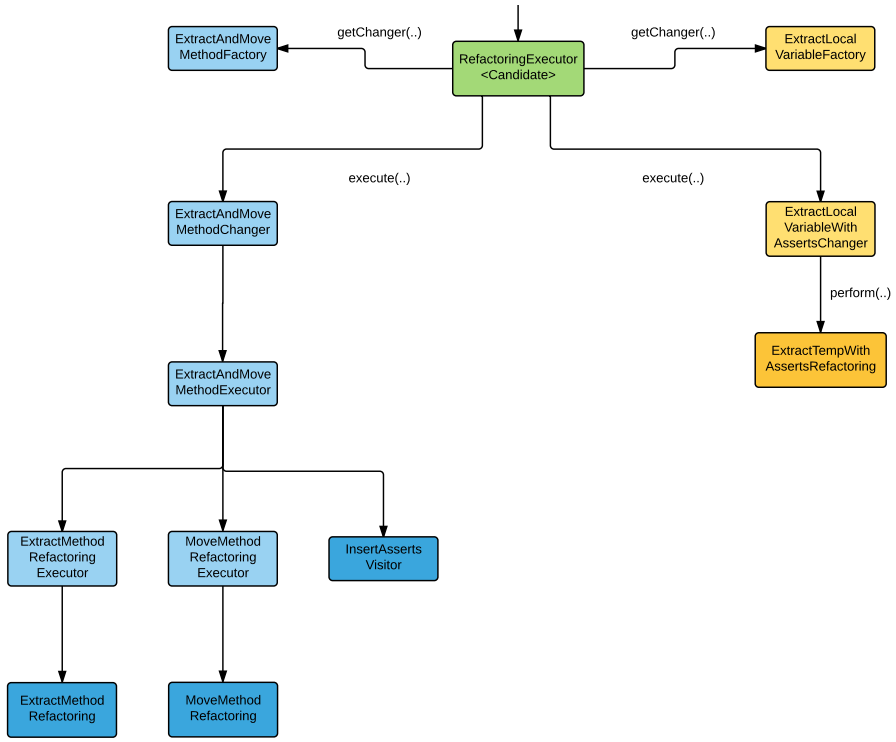

Figure 5.2: Execution diagram of the refactorings. The generic class `RefactoringExecutor` fetches the changer (blue for Extract And Move Method, yellow for Extract Local) from the factory objects, passes the arguments from the analyzer or the user to them, and the executions splits up into the refactoring specific executor classes. The four leaf classes at the bottom (slightly darker color) is responsible for creating the source code changes.

The refactorings themselves are invoked by a generic `RefactoringExecutor`, and instantiated through the factory object. The refactoring arguments and the handlers for the Java elements are provided to the constructor. Extract And `Move Method` is executed in three steps: Extract Method, `Move Method` and Introduce Asserts, as apparent in the leftmost leaves in Figure 5.3. The asserts are introduced using the Visitor Pattern [11] on the moved method. Extract Local Variable is executed in one step with the asserts introduced during the refactoring. Although the generation of the predicates are very similar, the locations we insert them and the implementation for that, is very different. The problem of inserting the asserts in the optimal place is hard, and is discussed in Section 4.1.3. In our implementation we prioritize the syntactical correctness of the resulting method.

In the following two sections we give an account of the implementation of the two refactorings and of the dynamic checks.

5.5 Extract And Move Method

In this section we describe the implementation of the refactoring defined in Section 2.7 and the dynamic checks from Chapter 4. Apart from the assert generation, some documentation and code layout, the implementation of this refactoring has been kept as it was in Kristiansen's work [17]. The execution is shown in Figure 5.3.

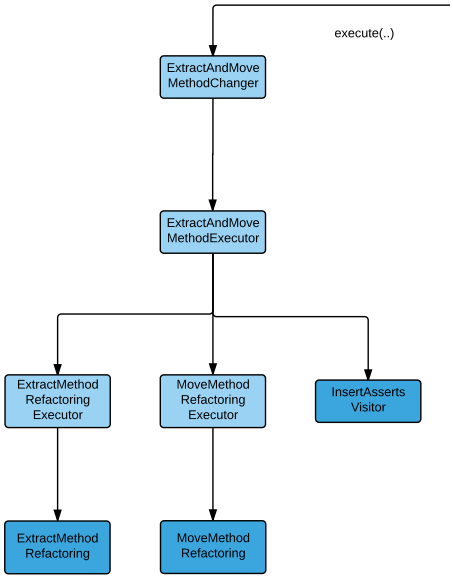

Figure 5.3: Extract and Move caption. The full diagram is found in Figure 5.6.

The implementation of `Extract And Move Method` is composed of two reimplementations of the Eclipse JDT refactorings, invoked sequentially from `ExtractAndMoveMethodExecutor`, where the resulting method of `Extract Method` is provided to Move Method. This omits the need to look it up from the compilation unit again and reduces run time. A similar approach could have been used for the assert generation, but to get a working prototype we used the slower solution of looking up the moved method by its name (unique by generation) and class to get the new method declaration ASTNode. How to do this is explained in Section 3.2.3. Looking it up on name works since the generated name is unique, but is not an ideal solution, and does not work if the target type is in the `default` package, due to the Eclipse JDT code. We do this lookup to avoid developing the infrastructure to collect and pass the result of the previous execution step, which is complicated, since the Eclipse refactorings are not particularly well suited for composition [17]. We then pass the visitor to the method's ASTNode and walk the subtree.

5.5.1 Implementing assert generation

In this refactoring we implement one of our early ideas for a check locations heuristic. We insert an assert statement, generated as described in Section 4.2 *before every method invocation except the first in the method*. To describe the algorithm we use the names from Listing 21. The purpose of this simplified, ideal example is to give names to the elements we need to talk about assert generation.

Before

```
1  class C{
2      public E e;
3
4      public void f(){
5          ...
6      }
7  }
8  class E{
9
10 }
```

After Extract And Move

```
class C{
    public E e;

    public void f(){
        e.h(this);
    }
}
class E{
    public void h(C c){
        ...
    }
}
```

Listing 21: Extract and `Move Method` working example, to remind us what the refactoring do and of the names we use throughout this thesis.

We have a class C with a public field e of type E and a method f. We omit other methods belonging to E, and show the code in f's method body as dots. They are highlighted to show that they are selected to be extracted and moved with e as target. In practice there will most likely be unselected code left in f, but we simplify. On the right side the selected code in f's body is replaced with a method call to the new method h, in E, and we pass a reference to `this` to the parameter c).

For Extract And Move Method, the assert generation is performed on the already extracted and moved method, here named h. The assert generation algorithm needs the target of the refactoring, e, the introduced variable c, and the ASTNode representing h's method declaration. A simplified version of the

algorithm is given in Algorithm 6.

input : m method
input : c the name of the variable of type C
input : t target field of class C, accessible from m through c
output: m with dynamic checks added

```
1 S ← method body of m, as a list of statement;
2 assertStatement ← assert c.t == this;
3 for s ∈ S do
4 |   if s contains a method invocation then
5 |   |   insert assertStatement in S just before s;
6 |   end
7 end
8 remove consecutive assertStatements in S;
```

Algorithm 6: Assert insertion algorithm for Extract And Move Method

Generating the predicate. To generate the assert we need to define the predicate: `c.e == this` and wrap it in an assert statement: `assert c.e == this;`. The algorithm needs the name of e passed as an argument, with the syntactic precondition that it is a member of C In practice however, this is already checked by the refactoring itself. The target is then prefixed with the introduced variable c's name. We add an error message to the assert statement; this is helpful for the programmer in searching for these asserts in the source code (simply searching for assert statements can give false positives, and the programmer might not know the predicate himself) and to communicate why the code is broken and which refactoring did it: `assert c.e == this : "Extract And Move Method";`.

Insert location. To find the insert locations we walk the method declaration AST and look for method invocations. A simplified example of such a tree is shown in Figure 5.4. Notice that the first parameter is the generated one: in our implementation this will always be first if it is present. However, we do not force the creation of this parameter, so in the cases where the refactoring did not generate it (because no member is accessed directly from the resulting method, or they are accessed in another way) we do not have a way of referring to the target, and we do not generate assert statements. This is discussed further in Section 7.

When we find a method invocation, we generate an assert statement and insert it *as the last statement before the one containing the method invocation.* The reasoning for this, as well as alternative approaches, is explained in Section 4.1.3. We walk the AST, using the Visitor Pattern [11] defined in the `InsertAssertVisitor` class extending the ASTNode. To insert a statement in a specific location in the method body, we must use a `ListRewrite`, as explained in Section 3.2. We specify the location to the `ListRewrite` by giving it a reference to the statement we want to place our assert in front of, i.e. the statement we found the method invocation in. Note that the ListRewrite operates on the *list of Statements in the method declaration body*, but we walk the

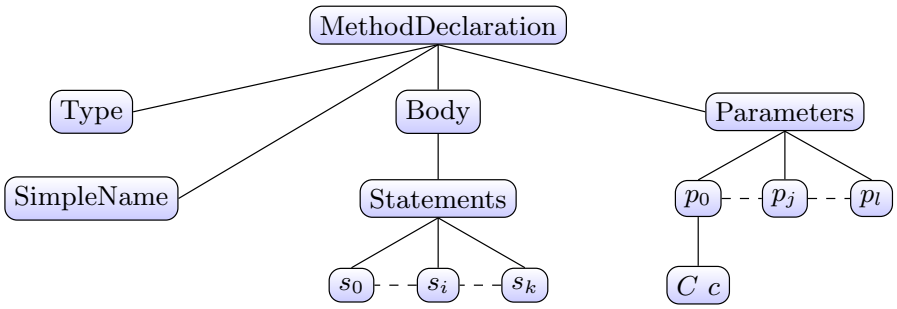

Figure 5.4: Simplified Eclipse AST representation of an extracted and moved method. The first parameter will be our the one used for the dynamic checks as described in Chapter 4

full subtree of the method declaration node: when we visit a statement node it will not necessarily be present in the list of statements; it can be a node further down the AST. Therefore, when we visit a method invocation, before we can use that statement as a location for the assert statement to be put in front of, we must loop over the node's parents until we find a statement that is in the method body list, as shown in Listing 22.

InsertAssertsVisitor

```
1 ASTNode getLocationStatementFromFragment(ASTNode statement) {
2             while(!statementList.contains(statement)){
3                     statement = statement.getParent();
4             }
5             return statement;
6 }
```

Listing 22: Searching for the right level in the AST: the `ListRewrite` needs a reference to a statement in the method body to know where to place our assert statement.

Heuristic Evaluation

The heuristic used here, of looking for method invocations, was an early idea – the first we implemented – and has several drawbacks. Method invocations are not the program elements that produces the behavior change: reference evaluation is; and while method invocations cover some occurrences of the `this` keyword (member method invocations, this or members passed as argument, etc) it does not account for all references to *this*, neither does it exclude what we point out in Chapter 4, that an intermediate reassignment might be acceptable. The latter is not a problem, as it indicates that er assert a stronger property, but we point out that this is one source of error in our experiment. The same goes for the assert statements introduced in the statement list of the method body: introducing a statement in the *innermost statement list node* would be an improvement, and the wrapper methods of all `this` references would be ideal,

although that would require us to change all implicit reference to `this` to be explicit.

5.6 Extract Local Variable

This refactoring is a reimplemented version of the Eclipse JDT's Extract Temp refactoring, as described in Section 3.3.4 to correspond to the `Extract Local Variable` definition given in Section 2.4. The classes responsible for this refactoring are shown in Figure 5.5: the leaf class is a copy of Eclipse's Extract Temp class, and the changer checks the preconditions and adds the change to the undo manager. We have made two changes to Eclipse's implementation: we have added the option to specify in which selection all occurrences will be replaced, and we have added assert generation for every replaced expression. We will describe each of these modifications in detail.

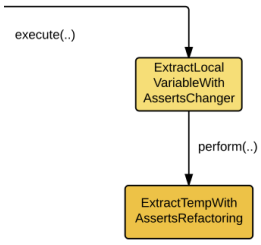

Figure 5.5: The `Extract Local Variable` refactoring classes. The full diagram is found in Figure 5.6.

5.6.1 Implementing assert generation

In this refactoring we use the method described in 4 for inserting one check for every substitution made. We do this in the same step as the substitution happens, so every time `Extract Local Variable` replaces a reference to the expression with a reference to the newly declared variable, we insert an assert statement as the previous statement. To comfortably describe the algorithm, we will refer to the simple code example shown in Listing 23.

Before

```
1 public void f(){
2     e.m(this);
3     e.n(e);
4 }
```

After Extract And Move

```
public void f(){
    E temp = e;
    temp.m(this);
    assert temp == e;
    temp.n(temp);
}
```

Listing 23: Extract Local Variable with Asserts example. Here the whole body is taken as the selection argument.

We have a method f, an expression e and a selection S (not shown) which is taken to be the whole method body. A simplified version of the modified refactoring algorithm is given in Algorithm 7. Note that this is the full `Extract Local Variable with Asserts` algorithm, as we modified it to generate asserts instead of producing them as a separate step.

input : e: an expression of non-void type E
input : S: a selection, as a list of consecutive statements
input : *context*: the outermost, non-type scope containing S
output: *context* with e safely extracted to a local variable in S
1 $v \leftarrow$ fresh variable name;
2 $assertStatement \leftarrow$ assert e == v;
3 $V \leftarrow$ all occurrences of e in S;
4 **for** $e2 \in V$ **do**
5 | replace the occurrences of $e2$ with v;
6 | insert $assertStatement$ in S just before the statement containing $e2$;
7 **end**
8 remove consecutive $assertStatement$s in S;
9 add a new variable declaration `E v = e;` to *context* just before S;

Algorithm 7: Modified `Extract Local Variable` to generate and insert asserts

To implement this algorithm we made two modifications to the refactoring implementation. We had to generate the predicate, which were done in the same way as for `Extract Method`, except we defined the following predicate `e == v` where e is the expression argument and v is the extracted variable. We wrapped it in the appropriate types and added a refactoring-specific error message. The insert code closely resembles the code for `Extract And Move Method` in Section 5.5. However, instead of walking the tree, we operate on the list of `ExpressionFragments` to be replaced, and every time one is replaced, we iterate over the parents of its node to find a selection in the method body list, and insert our assert as the previous statement. Finally we iterate over the list and remove consecutive asserts statements to remove clutter in the code.

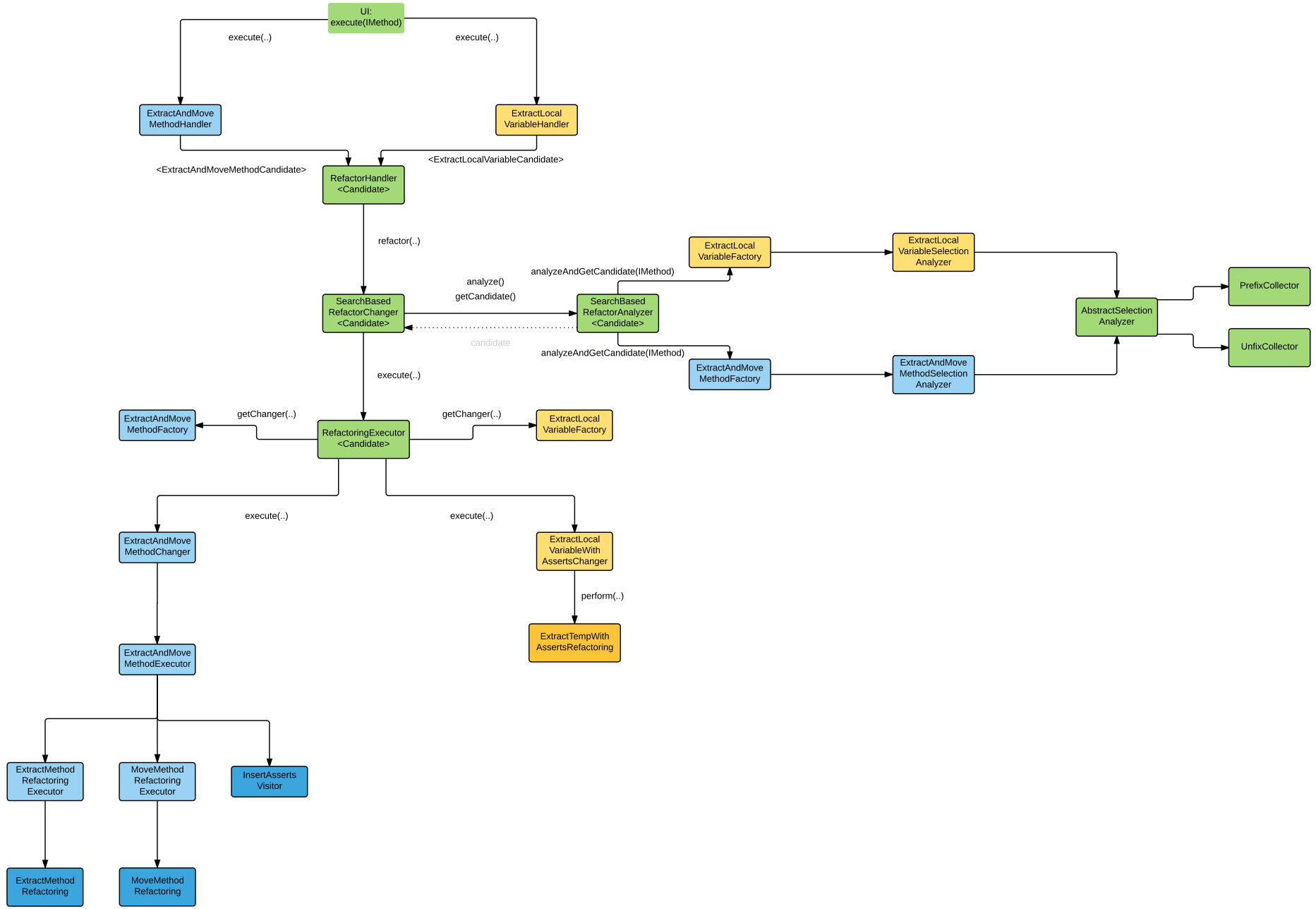

Figure 5.6: The key parts of the execution path when either refactoring is invoked on a method. Framed boxes represent classes, the UI box represents the user. An edge from a box to another indicates the pointed-to class being called by the pointed-from class. Edges are denoted with methods or generic arguments when appropriate. Green classes are generic and used by both refactorings, blue classes are solely for `Extract And Move`, and yellow for `Extract Local`. All classes are from our plugin. Calls to Eclipse's classes are not shown. The bottom leaves (all non-collector leaves) are constructing changes to the source code.

Chapter 6

Experiment

In Chapter 4 we gave examples of Java structures where `Extract Local Variable` and `Move Method` (defined in Section 2.4-2.7) change the program's behavior if they are applied without runtime precondition checks. We formulated a general substitution property whose violation predicts the observed behavior change and proposed how it could be formulated as a precondition and integrated in a refactoring tool for Java. In Chapter 5 we describe the implementation of a proof-of-concept refactoring tool, a plug-in for Eclipse providing a UI for invoking `Extract Local Variable` and `Extract And Move Method` on program elements and perform the refactorings with integrated dynamic checks. The plug-in can be invoked on high-level program elements, like projects, and will perform the selected refactoring on all possible methods (decided by a heuristic). In this chapter we describe the experiment we performed to verify the practical benefits of our proposed solution. We report, and discuss, the results of our case study on a large, well-known, open-source software project in Java.

6.1 Experiment design

The purpose of this experiment is to verify if the program structures we previously discussed exist in real-life code and can be refactored with resulting behavior changes. We also evaluate whether the dynamic checks we propose are useful and capture such changes. A case study is ideal at this point because we are early in the development of our solution, and a qualitative study can provide a good assessment of different – perhaps overlooked – aspects of the evaluated idea. The study is quantitative over methods: we perform a large quantity of the refactorings (on each method with a possible target and location); however it is performed on one specific software project. Other alternatives for experiment design is given and evaluated in Section 6.3.

What we aim to learn from the case study is whether such program structures can be expected to appear "in the wild", and if so, if our asserts would prove to be useful. To validate our idea we execute a large number of our refactorings with dynamic checks on "real" code and see if the introduced asserts will alert us about any behavior change. We use a heuristic (described in Section 5.3) to apply the refactorings, an automated tool for applying them with integrated

dynamic checks (described in Section 5.4), run this from an Eclipse plug-in. The implementation, and the work it builds on, is described in Chapter 5.

When running our experiment we are interested in:

1. can we introduce dynamic tests without breaking the code?
2. do the tests trigger any of the generated assertions in the refactored code?
3. are the triggered asserts *sound*, i.e. do they tell us about actual behavior changes resulting from the refactorings?
4. are the triggered asserts *complete*, i.e. are there behavior changes that are not captured by them (but by the tests)?

6.1.1 The test case

When picking a test case, we considered the following properties:

1. Software Language: our focus is the Java programming language, so the code should be written in Java.
2. Size of the code base: a large code base could increase the amount of applied refactorings applied and heighten the probability of an occurrence of the kind of code structure that introduce the behavior change.
3. Coding style: the level of object-orientation, length of methods, complexity of prefixes, and the the overall coding style of the developers of the project will affect the results. We reason that a high level of object-orientation is necessary for our heuristic to find possible applications of the refactorings. Apart from that we consider variety of style an important property of a case project.
4. High quality, high coverage, test suite. Asserts are checked at run time, thus we depend on the project having an automated test suite with high coverage of the code.

We applied the heuristic and the refactorings to the Eclipse JDT UI Project, described in Section 3.2. We believe that this project is a good representative of professionally written Java source code, where the number of contributors contributes to its validity as object of a case study. It comprises over 300.000 lines of code (excluding blanks and comments), with more than 25.000 methods, and has an extensive set of unit tests. The test suite we used is the Automated Test suite for the Eclipse JDT UI project, which we ran using the JUnit workbench in the Eclipse IDE. It is a completely automated test suite, [1]containing 2396 test cases. This was also the test case of a related case study by a previous master student [17] and we could compare our results to his, as well as use his tool as a starting point for our development work (as described in Section 5.1).

We also tried the Apache Math Library as input, but that produced very few applications of the refactorings (300 applications of `Extract Local Variable`), and the only conclusion we were able to draw is that the code style of this project was poorly fitted for our use.

[1]org.eclipse.jdt.ui.tests.AutomatedSuite.Java

6.1.2 Set-up

In an Eclipse-instance with our plugin, we imported the Eclipse JDT UI code for versions 4.5 (with all dependencies) and the corresponding tests. Before the refactorings were invoked on the code, we ran the Automated Test Suite, and found that all unit tests passed. Before the experiment Eclipse reported two errors of the project, related to Java build version and project settings. We fixed these errors manually. We invoked the `Extract Local Variable` refactoring on the whole project, and ran the tests on the resulting code, recording the results. We then checked out the original code again, and invoked the `Extract And Move Method` refactoring on the unrefactored code. Then we ran the tests on the resulting code. We did not refactoring already refactored code, nor the test code itself.

To find the number of generated methods we search in the project for the string "generated_" for `Extract And Move Method`, or "temp_" for `Extract Local Variable`. As described in Chapter 5 this is the form we use for generating names. We search before and after running the refactoring, and subtract the first result from the last. This method was chosen due to problems concerning the logging of refactorings where they were stopped due to problems with the assert generation or bugs encountered when writing changes to the AST. Similarly, we count introduced asserts by searching for the phrase "due to aliasing" – part of the generated error message, and a phrase not used anywhere else in the project. We include extensive logging in the code, but this proved to be a more reliable way of counting the source code changes.

6.2 Results

In this section we report the results of our experiment.

6.2.1 Extract Local Variable

Invoking the project-wide `Extract Local Variable` refactoring on the full Eclipse project resulted in 4 538 single refactorings and 7 665 assertions. The refactoring introduced no compile errors. We then ran the Eclipse JDT UI Automated Test suite on the refactored code. The test suite finished with 4 failures and 11 errors. The difference between a failure and error in this case, is whether the test expected an exception or error, or not. The 4 failures originated from violation of our generated asserts. The 11 errors were due to build issues, where the build file required an old version of Java that did not handle our generated asserts, and consequently one file did not finish building. Changing the target Java version in the build file resolved the build problem and removed these 11 errors, which left us with only the 4 failures. In addition we had 133 violations of the generated asserts that were reported in the console output from the tests, but did not seem to affect the test results. Running the test suite without asserts produced no failures and no errors (after modifying said build file).

Assert violations The reported assertion violations originated from two specific asserts. In both cases the extracted expression was a get-method; in one case it seemed to be a factory-method while in the other the assert was triggered

Get-method

```
1       public String getName(){
2           return "name" + id;
3       }
```

Listing 24: Example of the type of get-method for strings that introduced the behavior change.

	Extract and Move Method	Extract Local Variable
Executed refactorings	755	4538
Generated asserts	610	7665
Resulting compile errors	14	0
Tests failing before	0	0
Tests errors	84	11
Tests failures	161	4
Asserts triggered in tests	0	2
Instances of asserts triggered	0	137

Table 6.1: These are the results of our experiment

by a method returning a string as illustrated in Listing 24. The string-object is created in the getter instead of accessing a field or otherwise stored reference. Calling such a method twice will produce objects that may evaluate to equal using the `equal`-method, but will not not be reference-equal as expressed with `==`.

6.2.2 Extract And Move Method

We invoked the Search-Based `Extract And Move Method` refactoring on the full (unrefactored) Eclipse project, resulting in 755 applied refactorings and 610 assertions. This produced 14 compile errors, excluding the two project-setting related errors we had from before. The first time we applied our refactoring we got 180 compile errors, and we incrementally improved our heuristic to exclude targets that would introduce the different types of errors, as explained in Chapter 5. 3 of the 14 compile errors were due to project specific settings (e.g. error on unused imports). Most compile errors were due to references to non-visible or unaccessible members, and missing imports, and one case due to project-specific settings of error on dead code (leading to the null-pointer precondition introduced in Section 2.7). Running the Automated Test suite on the resulting code (with compile errors) produced 84 errors and 161 failures. The difference between errors and failures are the same as already explained for `Extract Local Variable`. No asserts were found violated. Manually correcting all compile errors (as best we could) and rerunning the tests produced no errors or failures, and still no assert violations. Thus, we did not sift through the original test errors and failures with the intention of cataloguing their source.

The results are summarized in Table 6.1.

We should point out that for `Extract And Move Method` we still had some refactorings that were executed but without generated asserts. Our tool aborted

the insertion of asserts if it was clear (usually due to visibility issues) that the asserts would produce a syntactically incorrect program. We did not keep a history of the method-level changes in the refactoring, and did not undo the ones where the algorithm found it impossible to generate asserts. This means that we are only applying the runtime check at a fraction of our `Extract And Move Method` refactorings. In future work we could like to introduce special get-methods for these cases, as described in Section 4.2. Another approach would be to increase visibility of referenced fields, but this would require yet another check of correctness.

6.3 Discussion

A discussion of the experiment setup and results is found in Chapter 7.

6.4 Experiment Conclusion

We conclude that our experiment was well suited for this point in the development of our ideas, and we would like to repeat it with an improved version of the refactoring tool for more code bases. The implementation of the assert generation is not yet ideal, as our results tell us. Nonetheless, the results are promising and there are many improvements that can be done. For a discussion and conclusion of the ideas and results in this thesis, look to Chapter 7-8.

Chapter 7

Discussion

In this chapter we discuss the decisions and results from the previous chapters and describe future and related work.

7.1 Topic motivation

This thesis is motivated by the observation that common refactorings can easily, and accidentally, change a program's behavior. We have inspected corner cases for the common refactorings described in Chapter 2 for Java that introduce a particular type of behavior change. We have formulated a property whose presence we believe indicates that the behavior change will not occur, a way of encoding it in Java, and aim to verify this by running a case study experiment.

The refactorings. We are aware that the corner cases are very particular, and that our assessment of them can be biased by our position as refactoring tool developers [22]. This is why we aim to verify that they do indeed exist in a "wild" source code, by implementing an automated tool that can apply the refactorings with checks to a large code base. However, we do agree with Steinmann's stance, that bugs in refactoring tools should not be excused but removed [4].

`Extract Local Variable` is a commonly implemented and used refactoring, and thus we think is important to guarantee correctness of. It is implemented in IDEs of several languages, and indeed the behavior change we look at can be introduced by a similar operation in other, object-oriented languages. `Extract And Move Method` is not a commonly implemented refactoring, but is composed by two common refactorings and can be seen as a particular case of `Move Method`. Precomposing `Move Method` with `Extract Method` highlights the similarities between this refactoring and `Extract Local Variable`. The two refactorings have similar precondition checking problems with a similar solution, and `Extract Local Variable` can be used as a precondition check for the more complex refactoring.

Another approach to implementing assert generation for `Extract And Move Method` uses this similarity. We can first perform `Extract Local Variable` with assert generation, and then `Extract And Move Method` with the extracted variable as target. This would result the references in the assert statements sub-

stituted as in the rest of the code, and we would not need the assert generation as a separate step. In our implementation this was not considered an option, as the Eclipse refactorings are challenging to compose [17].

Behavior change. Calling something a bug in a refactoring invites the discussion on the definition of behavior and refactoring specifications that we visited in Section 2.3. We aim to not take too much of a stand in regard to behavior and motivate our asserts in the code exactly like this. Until the agreement of a definition of behavior, the refactoring tool cannot check if behavior is preserved. Thus its responsibility should be considered to effectively communicate to the programmer what possible behavior changes could have happened, and while this is currently done by the use of previews [9], we believe our assert statements have the advantage of being a runnable contribution to an existing test suite and a possibly lasting change in the code. The latter means that it could serve as a documentation step of which refactoring had been applied to which arguments and what semantic *risks* it introduced. Adding to the test cases by instrumenting the code also means that we can inspect not only the observable behavior, but also the inner program structures. Soares et al. [29] have generated test cases for finding transformations that introduced behavior change, but we see our white-box testing (a test of the internal structure of software) as an improvement.

Asserts. The asserts we use to check the precondition can be seen as clutter, and as an unsatisfactory addition to source code. However, they do effectively communicate to the programmer the preconditions, and possible violations, and they allow the check of a preconditions that is impossible to guarantee by static analysis. Refactoring tools should provide comprehensive error messages [9], communicating *what* is wrong, and why. We believe that a clickable list of the resulting asserts (future work), where in the code they are found and why, could provide the programmer with enough information to decide herself whether the refactoring is correctly applied, and might even make running the code unnecessary.

An alternative to using Java's assertions is JML [18]. This would give the advantage that the assertions would not introduce clutter in the source code (only the comments), and the additional state-keeping would be confined to JML ghost variables. This could be combined with a custom IDE where the checks were either not visible to the developer at all, or only visible in a special Refactoring View or similar. A drawback with a separate view is that it interrupts the programming from his regular activities and obscures the implications of their refactorings from them. Finding a way of implementing such check and effective communication with a user-friendly interface is considered future work.

Compiler optimization. The problem we have are somewhat similar to one in compiler optimization. An optimization effect can be gained from minimizing redundant loads of fields by using the previously loaded value if it has not changed in-between. This analysis suffer from the same limitations and the refactorings we have considered. The compiler optimization this technique is known as Common Subexpression Elimination, and is done statically. On the JVM-level this has been tackled under the name "Hot Field-analysis" by Wim-

mer et al. [31] through a dynamic technique that efficiently switches back to the unoptimized and correct behavior if it detects that the relevant heap has been modified. This can be compared to using static analysis to determine all allowed substitution locations, and for the ones that cannot be determined in this manner, use the original – possibly more complex, in the case of `Move Method` – qualifier.

7.1.1 Specification discussion

We are concerned by the lack of refactoring specifications. As mentioned in Chapter 2 both Opdyke and Fowler have attempted refactoring catalogues, but neither can currently serve that purpose. Schäfer et al. [27] give a concise, formal definition of some refactorings that they can translate easily into code for the JastAdd [8] attribute grammar framework for Java. For the refactorings they look at, they are mostly concerned with visibility and shadowing, and consequently make use of infrastructure that tracks such references and either keeps bindings consistent, or rejects a refactoring if the refactored program would have different bindings. Graph transformations have also been used to specify refactorings by Mens et al. [20]; however, in the particular case of the Move Method refactoring, they have opted to only deal with static methods/calls. Also Ó Cinnéide's "minitransformations" preserve behavior due to a restriction to structural manipulation [5].

A common challenge in specifying refactorings for common languages is the low "refactorability" of the languages. [4]. Java is a complex, but commonly used language and it provides challenging when it comes to defining refactorings. One could try to approach a refactoring catalogue for Java by starting with a subset of the language (like Featherweight Java [13]), try to define refactorings for this ideal subset, and then generalize them for the rest of the language. The objection is of course that the generalizing can be very hard.

7.2 Experiment Discussion

The experiment set-up was discussed in Chapter 6. We have evaluated our approach by refactoring a code base in the same way that we anticipate developers would do. We execute existing unit tests and observe if the generated assertions are triggered, which would indicate that the refactoring indeed changed the behavior.

Our findings show a limited success with some triggered assertions. We conclude that a developer *may* accidentally apply the refactoring incorrectly. However, our experimental setup yields a low number of applied refactorings and generated assertions.

7.2.1 Experiment design

For our experiment we chose a case study in which we used a heuristic and an automated refactoring tool to generate data on behavior change and practical value of our dynamic test idea. There are other experiment set-ups we could have used.

Traditionally similar research is carried out by analyzing refactoring data produced by programmers [30, 21, 22]. Data about used refactorings in Eclipse exists [22], but we needed to connect the refactorings to source code and be able to run the refactored code, which is not an option with this data set. A more feasible approach is to collect the data by mining repositories: we could identify where in the history of a public source control repository one of our supported refactorings had been applied, add our assertions and run the code to look for changed behavior. This would be useful as it would also capture manual refactorings, but it would require a very different setup and longer time than we had available. Mining software repositories for refactoring has been discussed (and criticized for being unreliable) [22].

Another is to conduct experiments with programmers, asking them to refactor code during an experiment session or in their regular programming activities. By tracking their regular activities we would get a less biased outcome than what have been produces by some research in regulated environments: programmers' knowledge about refactoring vary greatly [15], and many try to avoid refactoring code because of this [15]. Thus, when they are asked to refactor code in an experiment setup they might behave very different from when they are working. Participation is also a limiting factor: often this research is carried out on students who might not be representative for all programmers. In any case, this is a very different setup from what we chose, and although we think it would be very interesting to see how developers would use our tool in their daily refactoring activities, we consider this outside the scope of this work.

We opted for a more automated approach, and used an experiment setup where we could generate our own data. Along with an automated refactoring tool we needed a large code base to refactor. Variety in coding style, level of object orientation, method length, and how "refactored" the code already is would impact our findings. To increase data quantity and variation we could have done this quantitatively over many projects. However, setup of projects and tests takes time, and the tools was gradually improved throughout the experiment runs, before the final run, so we did not have time to try many projects. We anticipated this, and decided early that this work would report on a case study. While case studies always run the risk of the case not being representative for the whole domain, they can often allow more close examination of the findings and can lead to insights and surprises that can steer the research in new directions. As such, a case study is especially valuable early in the research process, and this one lead us to important insights like the extension of the test case catalogue (included in the GIT repository) and the idea of wrapper methods (described in Chapter 4).

7.2.2 Threats to validity.

The following issues have to be kept in mind when considering the experimental results:

- The implemented strategies for inserting the dynamic checks are heuristics. In Chapter 4 we describe the difference between introducing local assert statements in the refactored scope and wrapping the references in assert methods. In our development, described in Chapter 5, we use direct assert statements introduced as an element in the statement list constituting the

refactored method's body. This means that several statements could have been evaluated between our assert property and the reference we should check. An improvement would be to introduce the checks as an element in the innermost statement list node of the reference.

- The implemented strategy for looking for locations to insert is a heuristic for `Extract And Move Method`. As described in the implementation, we look for method invocations, which were our initial idea. This works for simple examples, because a method invocation will often entail an implicit `this` reference, which does not often occur in a "behavior-inducing" way without a method invocation. It could, however, do that – for example by accessing a field and throwing an exception, or by using a null-check of a field in a control flow.

- The number of identified instances where the `Extract And Move Method` refactoring can be applied depends on the coding style of the code base. A "perfectly refactored" project, or a project using less object-orientation, will have a lower number of possible instances, which is illustrated by the limited success we had with the Apache library. This is one of the reasons why an experiment where we track programmer's refactorings during development would be beneficial: the test case we consider is a "finished" code, and refactorings are often performed on unfinished code which can look very different from code in production.

- As described above, we had applications of `Extract And Move Method` where we could not generate assertions due to issues of field-visibility. This lowers the potential for assertions to be triggered (although changed results could still be uncovered by failing unit tests).

- Our evaluation uses unit tests to detect changed behavior. Our results depend on the coverage of the test suite. We did not check the test coverage due to time constraints.

- The total number of executed `Extract And Move Method` refactorings with generated asserts is relatively low. It could be that we need a much higher number to statistically find such a case we are looking for.

Improving the number of results As pointed out in Chapter 5, if a target found by the analyzer makes the refactoring or assert generation impossible, a new target is not chosen. This is something we would have liked to introduce, but did not due to time constraints. Another improvement to the experiment code could have been to repeatedly apply the refactorings to each method until the code could not be refactored any further. This was also not pursued due to time constraints.

Alternative Implementation Tools As noted in previous works [17], the tools provided by Eclipse can be challenging to work with. Alternative tools that could be used instead are Rascal [16] and Nuthatch [2], both state-of-the-art tools for program transformation with support for pattern matching a technique Eclipse's AST API lacks support for.

The results Although the asserts did alert us of behavior changes, we did not consider the changes to be of high importance. In future work, we would like to extend our experiment to larger and more code bases. Additionally, in combination with repository mining, it would be interesting to identify where in the history of a public source control repository one of our supported refactorings has been applied, add our assertions, and see if we can discover any changed behavior.

Chapter 8

Conclusion

In Chapter 4 we describe our contribution of a dynamic version of a precondition check that includes external source code and structures that cannot be determined by static analysis. In Chapter 4 and 5 we describe how the precondition checking can be implemented in a tool. The proof-of-concept tool we developed for our experiment, described in Chapter 5 implements some of the discussed options. We describe the implementation of this plug-in and and tools used, assess its quality and suggest improvements. We describe the experiment setup in Chapter 6, and discuss threats to validity and alternative approaches. The tool logged possible behavior changes of the source code we refactored. We inspected the behavior changes, and rapport our findings in Chapter 6. We cannot confidently conclude whether the semantic changes could have caused a behavior change, but we do conclude that it is useful to detect what could be a behavior change.

Going back to our research question can refactorings be made safer by encoding preconditions as dynamic checks? This thesis shows that, yes, we have cases that can only be checked dynamically and we can insert assertions that perform those checks. But, our initial experiments indicate that there seem to be few cases where such assertions are actually triggered. We do however conclude our research question with

Yes, refactorings can be made safer by encoding preconditions as dynamic checks.

But we concede that further investigation is needed before such an approach is useful to developers.

The proof-of-concept source code is publicly available in the Git repository at `git://git.uio.no/ifi-stolz-refaktor.git` along with code examples and references to submitted bug reports.

Appendix A

Code examples

A.1 Behavior change code examples

A.1.1 Assignment and reassignment in one statement

In Chapter 4 we discuss the different ways of encoding the dynamic checks in Java. We discuss encoding the in the scope of the refactoring target, as `assert` statements and the alternative of generating assert methods that can be used as wrappers of the references.

In Listing 25 we show an application of `Extract Local Variable` where the extracted expression is assigned to, referred to, then reassigned, all in one statement. In the example we invoke the methods in the argument to an if-clause, but it might as well be a method call or another program construct. We show the result with assert-statements added in all possible places in the scope, and still they will not alert us to the behavior change (adding a main-method that invokes f on an instance of C will yield two hash code strings that are the same, implying n being invoked the same object twice, while inlining *temp* will produce two different strings, since the reference change in between the invocations). Listing 26 shows us the same application of the refactoring where the checks are implemented as generic methods wrapped around the reference. This solution is further discussed in Chapter 4.

Assert Wrapper

```
public class C {
    X x = new X();
    X oldX;

    void f(){
            X temp = x;
            assert temp == x;
            if(temp.n()
                && newX()
                && temp.n()
                && oldX()){
                assert temp == x;
            }
            assert temp == x;
    }

    private boolean oldX() {
            x = oldX;
            return true;
    }

    private boolean newX() {
            oldX = x;
            x = new X();
            return true;
    }
}
class X{
    public boolean n() {
            System.out.println(hashCode());
            return true;
    }
}
```

Listing 25: The result of applying `Extract Local Variable` with assert-statements added in all possible places in the scope to capture the behavior change that has happened, but the reassignment to a new value and then back to the old happen in-between two invocations of the asserts, but still such that it changes the outcome of the code. This code can be run by adding a main-method that invokes f on an instance of C, and inlining *temp* and deleting the asserts will produce the original code, and a different output.

Assert Wrapper

```
1  public class C {
2      X x = new X();
3      X oldX;
4
5      void f(){
6              X temp = x;
7              if(wrapAssert(temp, x, "Extract Local").n()
8                  && newX()
9                  && wrapAssert(temp, x, "Extract Local").n()
10                 && oldX()){
11             }
12     }
13
14     private <T> T wrapAssert(T temp, T t, String error){
15             assert temp == t : error;
16             return t;
17     }
18
19     private boolean oldX() {
20             x = oldX;
21             return true;
22     }
23
24     private boolean newX() {
25             oldX = x;
26             x = new X();
27             return true;
28     }
29 }
30 class X{
31     public boolean n() {
32             System.out.println(hashCode());
33             return true;
34     }
35 }
```

Listing 26: The same code as in Listing 26, but with the checks encoded as generic method wrappers instead of assert statements. Generic, because the method can then be reused for other expressions of other types in the same compilation unit. The evaluation of the assert statement can still be turned off and on, but the method *wrapAssert* will nonetheless be invoked during all runs.

A.1.2 Extract And Move Method with null-target

In Listing 27 we show an application of `Extract And Move Method` where the target is null: in addition to provoking a `NullPointerException` upon evaluation, the if-clause in the moved method now contains dead code and we effectively changed the semantic of the code from performing whatever was in the if-clause to throwing an exception. This refactoring is performed in Eclipse, and will yield this result in addition to a reference to the *C* object passed as an argument and not used. While the second problem is a simple bug in the refactoring code, the null-pointer problem is not as easily solved, and the user

should not have been permitted this refactoring.

Original code

```
public class C {
    X x;
    public void f() {
        if (this.x == null){
            //some code
        }
    }
}
class X(){
}
```

Extract Method

```
public class C {
    X x;
    public void f() {
       h();
    }
    public void h() {
        if (this.x == null){
            //some code
        }
    }
}
class X{
}
```

Move Method

```
public class C {
    X x;
    public void f() {
        x.h();
    }
}
class X{
    public void h() {
        if (this == null){
            //dead code
        }
    }
}
```

Listing 27: Example of `Extract And Move Method` with x as target and f's body as the selection argument. The resulting code has changed behavior from whatever was inside the if-clause to throwing an `NullPointerException` upon invoking the new method on x, if x evaluates to null.

Bibliography

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.130.

[2] A. H. Bagge and R. Lämmel. Walk your tree any way you want. In *6th Int'l Conf on Model Transformation (ICMT'13)*, volume 7909 of *LNCS*, pages 33–49. Springer, June 2013. doi: 10.1007/978-3-642-38883-5_3.

[3] K. Beck. *Extreme programming explained: Embrace change*. Addison-Wesley Professional, 2000. ISBN 0321278658.

[4] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software*, 32:80–83, 2015.

[5] M. Ó. Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance, ICSM 1999*, page 463. IEEE Computer Society, 1999. ISBN 0-7695-0016-1. doi: 10.1109/ICSM.1999.792644.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT symposium on the Foundations of Software Engineering (FSE)*, pages 185–194. ACM, 2007.

[7] A. M. Eilertsen, A. H. Bagge, and V. Stolz. Safer refactorings. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, LNCS. Springer, Oct 2016. To appear.

[8] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007. doi: 10.1016/j.scico.2007.02.003.

[9] S. Erb. A survey of software refactoring tools. Technical report, Baden-Württemberg Cooperative State University, Karlsruhe, Germany, May 2010.

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999. ISBN 0201485672.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1994. ISBN 0-201-63361-2.

[12] M. Hofmann and M. Pavlova. Elimination of ghost variables in program logics. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing, TGC 2007*, volume 4912 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007. ISBN 978-3-540-78662-7. doi: 10.1007/978-3-540-78663-4_1.

[13] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[14] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[15] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*, page 50. ACM, 2012.

[16] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *9th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177. IEEE CS, 2009.

[17] E. Kristiansen. Automated composition of refactorings. Master's thesis, Dept. of Informatics, University of Oslo, 2014. Available from `http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2014/kristiansen/`.

[18] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: Proc. of the 8th Intl. Conf. on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34. Springer, Nov. 2006.

[19] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[20] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6(3):269–285, 2007. ISSN 1619-1374. doi: 10.1007/s10270-006-0044-6.

[21] E. Murphy-Hill. Improving refactoring with alternate program views. Technical report, Portland State University, Portland, OR, 2006.

[22] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[23] W. F. Opdyke. Refactoring object-oriented frameworks. Technical Report GAX93-05645, University of Illinois at Urbana-Champaign, 1992.

[24] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997. ISSN 1096-9942. doi: 10.1002/(SICI)1096-9942(1997)3:4⟨253::AID-TAPO3⟩3.0.CO;2-T.

[25] B. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In G. Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003. ISBN 978-3-540-00904-7. doi: 10.1007/3-540-36579-6_10.

[26] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *12th Intl. Conf. on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003. doi: 10.1007/3-540-36579-6_10.

[27] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '10*, pages 286–301. ACM, 2010. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869485.

[28] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. ISSN 2325-1107. doi: 10.1561/2500000014.

[29] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE Software*, 27(4):52–57, 2010. ISSN 0740-7459. doi: 10.1109/MS.2010.63.

[30] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *34th International Conference on Software Engineering (ICSE 2012)*, pages 233–243. IEEE, 2012.

[31] C. Wimmer and H. Mössenböck. Automatic feedback-directed object inlining in the Java HotSpot virtual machine. In C. Krintz, S. Hand, and D. Tarditi, editors, *3rd Intl. Conf. on Virtual Execution Environments VEE*, pages 12–21. ACM, 2007. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254813.

[32] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported—An Eclipse case study. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 458–468. IEEE, 2006.