Integrating "Code Smells" Detection with Refactoring Tool Support

by

Kwankamol Nongpong

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Engineering

at

The University of Wisconsin-Milwaukee

August 2012

ABSTRACT

Integrating "Code Smells" Detection with Refactoring Tool Support

by

Kwankamol Nongpong

The University of Wisconsin-Milwaukee, 2012
Under the Supervision of Professor John Tang Boyland

Refactoring is a form of program transformation which preserves the semantics of the program. Refactoring frameworks for object-oriented programs were first introduced in 1992 by William Opdyke [73]. Few people apply refactoring in mainstream software development because it is time consuming and error-prone if done by hand. Since then, many refactoring tools have been developed but most of them do not have the capability of analyzing the program code and suggesting *which* and *where* refactorings should be applied. Previous work [49, 91, 19, 25] discusses many ways to detect refactoring candidates but such approaches are applied to a separate module. This work proposes an approach to integrate a "code smells" detector with a refactoring tool.

To the best of our knowledge, no work has established connections between refactoring and finding code smells in terms of program analysis. This work identifies some common analyses required in these two processes. Determining which analyses

are used in common allows us to reuse analysis information and avoid unnecessary recomputation which makes the approach more efficient. However, some code smells cannot be detected by using program analysis alone. In such cases, software metrics are adopted to help identify code smells. This work also introduces a novel metric for detecting "feature envy". It demonstrates that program analysis and software metrics can work well together.

A tool for Java programs called JCodeCanine has been developed using the discussed approach. JCodeCanine detects code smells within a program and proposes a list of refactorings that would help improve the internal software qualities. The programmer has an option whether to apply the suggested refactorings through a "quick fix". It supports the complete process allowing the programmer to maintain the structure of his software system as it evolves over time.

Our results show that restructuring the program while trying to preserve its behavior is feasible but is not easy to achieve without programmer's declared design intents. Code smells, in general, are hard to detect and false positives could be generated in our approach. Hence, every detected smell must be reviewed by the programmer. This finding confirms that the tool should not be completely automated. An semi-automated tool is best suited for this purpose.

ACKNOWLEDGMENTS

I would like to thank my major professor, Professor John Tang Boyland, for his insights and supports throughout the program. Professor Boyland has been so understanding and supportive through my ups and downs. Without his time and dedication, this work would not have been complete, concise and comprehensive.

I am thankful to the dissertation committee, Professor Ethan Munson, Dr. Adam Webber, Professor Mariam Zahedi and Professor Yi Ming Zou. Their valuable comments help cultivate this dissertation.

I thank William Retert who helped develop an analysis. I am also grateful to all members of the UWM's SLAP group for fruitful discussions on the research. Our weekly meetings have broadened my view to different research areas. I would also like to thank all undergraduate students who willingly donated their code to be used in testing and evaluating this work. Special thanks go to Songsak Channarukul and Tapanan Yeophantong for their encouraging words and suggestions.

Last but not least, I would like to thank my parents and relatives for their endless supports and encouragements. I also thank my sister who always takes my phone calls despite the time zone differences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter states the current problem in software development process as well as the motivation of this work. It further introduces the idea of this work and describes why the problem is not trivial and how we tackle the problem. The main contributions are also discussed in this chapter.

1.1 The Problem

A software system becomes harder to maintain as it evolves over a period of time. Its design becomes more complicated and difficult to understand; hence it is necessary to reorganize the code once in a while. The most important thing when reorganizing code is to make sure that the program behaves the same way as it did before the reorganization has taken place. Semantic preserving program transformations are known as *refactorings*. The idea of refactoring is first introduced by William Opdyke

in 1992 [73]. The behavior preservation criterion is also discussed in his work.

In the past, refactorings were not taken into the mainstream development process because applying refactorings by hand is error-prone and time consuming. The benefits of refactorings are not obvious to many developers because refactoring neither adds new features to the software nor improves any external software qualities. Therefore, many system developers give refactorings low priority. They are afraid that doing so would slow down the process and/or break their working code. Though refactoring does not help improve external software qualities, it helps improve internal software qualities such as reusability, maintainability and readability. It is inarguable that software design changes frequently during the development. Performing refactoring introduces a good coding discipline as it encourages reuse of existing code rather than rewriting new code from scratch.

Refactoring is usually initiated/invoked by the developer. Most software developers only refactor their code when it is really necessary because this process requires in-depth knowledge of the software system. While many experienced developers can recognize the pattern and know when to refactor, novice programmers may find this process very difficult.

Even with the knowledge of refactorings, it is not easy for the developer to determine which part of their code can benefit from refactorings. Many programmers learn from their experience. New generation programmers are more fortunate since Martin Fowler and Kent Beck address this issue in their book on Refactoring [33].

They provide a list of troubled code patterns which could be alleviated by refactorings. Such patterns are widely known as *code smells* or *bad smells*. Recent work by Mantyla and others [58] attempts to make Fowler's long monotonous list of smells more understandable. In their work, smells are classified into 7 different categories. The taxonomy helps recognize relationships between smells and make them more comprehensible to the developer.

Despite the presence of such guidelines, finding code smells is not trivial. First and foremost, the developer has to recognize those patterns. The problem is, even if he can recognize them, he may not realize it when he finds one. Such a task becomes much more difficult for a large scale software system.

The process of detecting and removing code smells with refactorings can be overwhelming. Without experience and knowledge of the design of the particular software, the risks of breaking the code and making the design worse are high. Applying refactoring carelessly can inadvertently change program behavior. When refactoring is carefully applied, we not only preserve the program behavior but also avoid introducing new bugs.

Many refactoring tools have been developed [44, 2, 86, 43, 24]. There are also a number of works on finding refactoring candidates [49, 81, 19]. Nonetheless, these two frameworks usually work separately. It is unfortunate to see two related frameworks work on their own and not utilize the benefits to their maximum potentials. The relationship between code smells and refactorings are obvious but not many people have put them together.

1.2 Proposed Solution

Code smells detection and refactoring are connected. While code smells represent design flaws in the software, refactoring is the process which restructures and transforms the software. In other words, code smells tell what the problems are and refactoring can then be used to correct such problems. Integrating these two processes would provide the complete process of locating the design flaws and improving software design. The integrated framework also provides other benefits which include:

1. *Clearer Connection between Smells and Refactorings*: It is evident that code smells and refactorings are related. However, the connections are abstract and usually obscured by their complexities. Putting them in the same framework presents their relationships in a more concrete way.

2. *Analysis Information Reuse*: Checking conditions before refactoring and detecting code smells require similar analyses (as discussed further in chapters 3 and 4). It is unnecessary to perform an analysis for information that we already have. Reusing analysis information makes the framework more efficient. However, to be able to correctly reuse the information, we have to keep track the parts of the program that change. Then, we must determine which analysis needs to be rerun to address those changes. The overhead of this framework will be keeping tracks of changes made on the code. I believe that such overhead is a small sacrifice for improved efficiency.

3. *Continuous Programming Flow*: With the combined framework, the developer

can check for code smells and remove them without disrupting the flow of their coding. It encourages the developer to make changes incrementally.

Some code smells introduce design change and require the developer's assistance. Not all code smells can be automatically detected. Hence, this work focuses only on those that can be detected automatically. A set of code smell detection analyses developed in this research is discussed in chapter 4.

1.3 Contributions

The major contributions of this research are:

1. It defines conditions that must be checked to ensure behavior preservation before refactoring.
2. It identifies analyses required for the condition check.
3. It identifies analyses required to detect code smells.
4. It shows relationships between code smells and refactorings (in terms of analysis used).
5. It introduces metrics to detect code smells.

Though this work implements a tool for Java programs, the theoretical ideas can be adapted not only to other object-oriented languages, but also to other programming language paradigms.

1.4 Outline of the Thesis

Related work is discussed in Chapter 2. In this chapter, we provide observations on current refactoring tools. Existing techniques for finding refactoring candidates are reviewed. Other analyses that could be used to ensure semantic preservation are mentioned in this chapter.

Chapter 3 discusses low-level and high-level refactorings, their complexities and their semantic preserving conditions. It explains the importance of semantic checks and shows examples of how careless refactoring could affect the observable behavior.

Chapter 4 describes each code smell and the approach that this work uses for smells detection. Refactorings that can be applied to remove smells are also discussed in this chapter.

In Chapter 5, we discuss some existing cohesion and coupling metrics and why they are unsuitable for feature envy detection. A novel metric to detect feature envy is introduced in this chapter.

Chapter 6 describes the overall framework of our implementation, *JCodeCanine* which is a tool that analyzes Java source code. It detects code smells discussed in chapter 4 and suggests a list of refactorings that could address the design flaws.

Chapter 7 provides discussion on empirical results. It looks at JCodeCanine's efficiency in various aspects including the comparison of code quality before and after smells detection and refactoring application.

Chapter 8 concludes the present work and some open problems for future work.

Appendix B presents a few case studies for code smells detection.

Chapter 2

Related Work

According to Opdyke [73], each refactoring basically consists of preconditions, mechanics and postconditions. All preconditions must be satisfied before applying refactoring. Likewise, all postconditions must be met after refactoring is applied[1]. These conditions ensure that the program behavior is preserved. Opdyke also categorizes refactorings into low-level and high-level refactorings. Low-level refactorings are related to changing a program entity (*e.g.,* create, move, delete). High-level refactorings are usually sequences of low-level refactorings. He also provides proofs of behavior preservation for many refactorings. The behavior preservation proofs of some low-level refactorings are trivial but implementing them is not as trivial. This issue will be discussed in Chapter 3.

[1]The use of term "postcondition" in this research is different from the standard use of postconditions. Here we have postconditions apply to perform checks that are difficult to do before the transformation takes place.

2.1 Refactoring Tools

In early 1990s, Don Roberts and his colleagues developed a refactoring tool called the Smalltalk Refactoring Browser [76]. This refactoring browser allows the user to perform many interesting refactorings automatically (*e.g.,* Rename, Extract/Inline Method, Add/Remove Parameter). However, this early tool was not popular because it was a stand-alone tool separate from the integrated development environment (IDE). Developers found it inconvenient to switch back and forth between the IDE (develop code) and Refactoring Browser (refactor code). Thus later refactoring tools have been integrated in the IDEs. The following are refactoring tools for Java.

IntelliJ IDEA [44] This is an expensive commercial IDE. This tool also supports Rename and Move Program Entities (*e.g.,* package, class, method, field), Change Method Signature, Extract Method, Inline Method, Introduce Variable, Introduce Field, Inline Local Variable, Extract Interface, Extract Superclass, Encapsulate Fields, Pull Up Members, Push Down Members and Replace Inheritance with Delegation.

RefactorIt RefactorIt is a commercial software that supports many automatic refactorings [2]. It can cooperate with Sun ONE Studio, Oracle 9i JDeveloper and Borland JBuilder. The supported refactorings are: Rename, Move Class, Move Method, Encapsulate Field, Create Factory Method, Extract Method, Extract Superclass/Interface, Minimize Access

Rights, Clean Imports, Create Constructor, Pull Up/Push Down Members.

JRefactory [86] JRefactory is a tool that is first developed by Chris Seguin. However, Mike Atkinson has taken over the leadership role since late 2002. This tool allows easy application of refactorings by providing user interface based on UML diagrams as visualization of Java classes. It can cooperate with JBuilder and Elixir IDEs. JRefactory supports the following refactorings: Move Class, Rename Class, Add an Abstract Superclass, Remove Class, Push Up Field, Pull Down Field, Move Method.

jFactor [43] jFactor for VisualAge Java is a commercial product that provide a set of refactorings. Extract Method, Rename Method Variables, Introduce Explaining Variable, Inline Temp, Inline Method, Rename Method, Pull Up/Push Down Method, Rename Field, Pull Up/Push Down Field, Encapsulate Field, Extract Superclass/Interface.

Transmogrify [85] Transmogrify is a Java source analysis and manipulation tool. This tool is under development and currently is focused on the refactoring tool. It is available as a plug-in for JBuilder and Forte4Java. It supports a limited set of refactorings such as Extract Method, Replace Temp with Query, Inline Temp, Pull Up Field.

Eclipse [24] Eclipse is a generic development environment by IBM. It also has refactoring support and some analysis. This project is open source. The current version of Eclipse (Helio version 3.6.1) supports many types of refactorings which include Move Method, Rename Method, Encapsulate Fields and etc.

Microsoft Visual Studio Microsoft Visual Studio is an integrated development environment. It supports many primitive refactorings.

Generally, refactoring tools provide a list of refactorings in which the user can choose from the menu. Once the user chooses which refactoring to apply, the tool performs analyses in the background checking the required conditions. If those conditions are met, then refactoring can be performed. A few tools, like the refactoring support in Eclipse, allow the user to preview the resulting code before committing changes to the code. The preview feature gives the user a better idea of which part of his code will be affected by such a refactoring and whether it corresponds with his intention.

There are many refactoring tools for other programming languages like C++. For instance, Xrefactory also known as xref [95] is a refactoring browser for Emacs, XEmacs and jEdit. CppRefactory [84] is another open source refactoring tool that automates the refactoring process in a C++ project. Though the refactoring framework was originally proposed for the object-oriented programming language, many researchers apply the idea of refactoring to other language paradigms as well. Li and his colleagues propose refactoring tool support for functional languages [56]. Saadeh and Kourie [79] developed a refactoring tool for Prolog. The general idea is similar

to that of object-oriented languages but the conditions and mechanics are different.

Although most tools discussed in this research apply refactorings by directly manipulating the source code, many software designers think about refactorings at the design level. Researchers who are interested in design-level transformations include Griswold and Bowdidge [40]. They state that it is rather difficult to conceptualize program structure by just observing the program text. Instead of using program text, a graphical representation of program structure is used as it permits direct manipulation of the program structure at design level. Gorp *et al.* [37], Enckevort [26] and Saadeh *et al.* [78] and propose techniques to apply refactorings to UML diagrams. We omit further discussion regarding design-level transformations, since our work uses source code manipulation approach.

Like Opdyke and Fowler, we believe that refactoring tool cannot be completely automated. A good refactoring tool should give the developers the final authority. It must interact with the developer because inferring design intent is difficult. Some refactorings may introduce a design change which requires software developer's insights because he has the best knowledge of the program context. A tool can facilitate the process by suggesting a set of refactorings and helps ensure that each refactoring is applied correctly.

2.2 Finding Refactoring Candidates or Code Smells

Kataoka *et al.* [49] propose that *program invariants* can be used to find refactoring candidates. First, they define patterns of invariants that identify a potential candidate

for each refactoring. If the program invariant match the pattern, it is considered a refactoring candidate. For instance, a parameter can be removed, if it is not used, it is a constant, or its value can be computed from other source. The invariant patterns p for Remove Parameter refactoring are:

- $p = constant$
- $p = f(a,\ b,\ \ldots)$

Their approach is independent of the technique to find invariants. Either dynamic or static analysis techniques can be used. The static approach requires the programmer to explicitly annotate his program with his design intent and most programmers consider it troublesome to carry out this task. Sometimes invariants are implicit. An alternative to expecting programmers to annotate code is to automatically infer invariants. Invariant inference can be done by performing statically [29] or dynamically [27]. With the dynamic approach, the program is instrumented to trace variables of interest. However, the results from dynamic approach depend on the quality of test suites. They are, in general, true only for a set of some program inputs. On the contrary, static analyses are sound with respect to all possible executions. Hence, it is desirable to combine static with dynamic approach but so far, no one has succeeded. A detailed evaluation of static and dynamic invariant inference tools is given by Nimmer and Ernst [70].

Kataoka's approach is applicable to a limited number of refactorings. Not all refactoring candidates are discoverable from invariants. Determining other refactoring

candidates requires other techniques. Generally, semantic analysis of the program is required.

Melton and Tempero [64] suggest an approach that identifies refactoring opportunities using dependency graphs. According to their statement, long cycles are hard to understand, test and reuse. If someone wants to understand a class in a cycle, he is required to understand every other class in the dependency cycle as well. Hence, cycles should be detected and removed. If a class is involved in many cycles, it is desirable to break the cycle by extracting an interface from such a class.

Many researchers use metrics to identify refactoring candidates [9, 81, 42, 80, 63, 82]. Bieman and Kang define cohesion as a degree to which modules belong together [9]. Simon and others [81] propose a *distance cohesion metric* which represents how close two or more program entities are. They use program visualization as an aid in interpretation of the results. The distance cohesion metric can be applied to a limited set of refactorings such as Move Method, Inline/Extract Class. Singh and Kahlon [82] propose metrics model to identify code smells which include cohesion and encapsulation metrics. They argue that encapsulation should be measured from the unity and the visibility of class members and introduce two new metrics for information hiding and encapsulation. In this work, a statistical analysis is applied on a set of software metrics which is then grouped by Mantyla's bad smell categorization [59]. The results show that their new encapsulation and information hiding metrics play a big role in identifying smelly classes.

Tourwè and Mens [91] introduce an approach that is independent of language syntax by using *logic meta-programming*. Similar to any logic programming, the key components of this approach are *facts* and *rules*. Each program has its own set of facts. Facts (*e.g.,* inheritance hierarchy) can be derived automatically from the code. Rules for finding refactoring candidates, on the other hand, must be defined manually. However, they are true and can be used for any program. This approach can be applied to any language; nonetheless, the results from this approach are in an intermediate form. They must be converted to another form so that they can be used easily in the successive steps. Such a conversion causes some overhead (mapping from program representation to facts and vice versa). The accuracy of results depends on the quality and the completeness of defined rules. Moreover, the intermediate representation may not map well back to the source. For instance, comments may be lost during the mapping.

While many works focus on detection algorithms for specific smells, Moha *et al.* propose a technique to detect design defects using high-level abstraction through UML class diagrams [65, 66]. They define a meta-model for defects' specification and algorithms to detect design defects from the meta-model. Their defect correction technique is based on a rule-based language.

A number of researches aim to achieve better accuracy in detecting code smells and/or finding refactoring candidates by performing both structural and semantic analyses [23, 21, 15, 72]. Conceptual relation or semantic information are extracted by information retrieval techniques (Shingles, Latent Semantic Indexing) and natural

language processing techniques. However, some semantic information can only be retrieved at run-time which makes semantic analysis (especially through dynamic metric) a lot more expensive than syntactic or structural-based analysis.

Unlike any previously discussed approaches that consider only the current version of the source code, a number of research works consider code history through change metrics [19, 80]. Demeyer *et al.* [19] focus on the reverse engineering effort by determining where the implementation has changed. They study the history of the software and find refactorings between successive versions of software based on *change metrics*. Three categories of metrics are considered: 1) method size, 2) class size and 3) inheritance. After deriving the metrics for each version of the software, numbers from those metrics are then compared. Any substantial differences imply that there were major changes between versions and that refactorings may have been applied. The cons to this approach is that software logs must be available; hence, unversioned software will not benefit from this approach. However, for versioned software, it helps understand design changes in a software system and learn how it evolves.

2.3 Code Smell Detection Tools

There are a number of tools that support automatic code inspection. The well-known C analyzer LINT [47] and its Java variant JLINT [3] can check for type violations, null references, array bounds errors, etc. These tools focus on improving code quality from a technical perspective. JDeodorant [93, 31] is another code smell detection tool which specifically identifies type-checking bad smells in Java source code. Code

smells that the early version of JDeodorant can detect Replace Conditional with Polymorphism and Replace Type Code with State/Strategy. JDeodorant, however, only identifies code smells. It does not provide any suggestions or recommendations on how such smells should be removed. Recent version of JDeodorant [31] also identifies God Classes, suggests where to apply Extract Class refactoring and allows the programmer to perform class extraction in the process.

The work by Eva van Emden and Leon Moonen [25] and RevJava [30] are more closely related to this work as they focus on improving code quality from a program design and programming practice perspective. RevJava is a Java analysis tool that performs design review and and architectural conformance checking. Based on predefined and user-defined design rules, the system analyzes Java bytecode, checks if any rules are violated and reports them to the user. However, RevJava is not suitable for large software systems because it has no support for visualization of rule violations.

Emden and Moonen [25] categorize code smells into two kinds of aspects: *primitive smell aspects* and *derived smell aspects*. Primitive smell aspects can be observed directly from the code. Derived smell aspects, on the other hand, are inferred from other aspects according to a set of inference rules. In this architecture, the detection of primitive smell aspects and the inference of derived smell aspects are treated as separate units, and consequently, they are extendable. The programmer can add new code smells by extending the inference rules.

Unlike RevJava, Emden's work analyzes Java source code because some coding standards cannot be checked on bytecode, since the bytecode contains less information

than the original source code. The code smells detection process is: 1) find entities of interest, 2) inspect them for primitive smells, 3) store information in a repository and 4) infer derived smells from the repository. After the smells have been detected, they are presented to the user by visualizing the source model using graphs.

2.4 Refactoring and Type Constraints

Type constraints are usually used to type check the program. Some researchers adopt this idea and add type constraints to the condition check in order to ensure behavior preservation in refactoring.

Tip *et al.* [90, 89] use type constraints for a set of refactorings that is related to generalization, *e.g.,* Pull Up Members and Extract Interface. The type constraints are used to verify the refactoring's preconditions.

Research work by Balaban *et al.* uses type constraints for library class migration [6]. All methods in legacy classes (*e.g.,* Hashtable and Vector) have been superseded by classes (HashMap and ArrayList respectively) which provide similar functionalities except they allow unsynchronized access to their elements. When replacing legacy classes with unsynchronized, *synchronization wrapper*. In their research, a set of type constraint rules to migrate program that uses legacy class are defined. The programmer must define a set of migration specifications. Type constraint rules are generated from the given program and migration specifications. The runtime of the analysis is exponential. They also provide an algorithm for Escape Analysis that is used to check for thread safety.

It is inarguable that type constraints is useful for many refactorings. However, its usage restricts to only refactorings that involve types and those that could introduce type violation if applied carelessly.

Chapter 3

Refactoring Complexities

Refactorings are structural changes made to a program that preserve program semantics. While many works [33, 88, 10] look at how refactoring could improve the structure and design of a program, this work focuses on the latter issue *i.e.,* how to check that the change is semantics preserving. Behavior preservation is important because if it is not assured, the program could produce different results after the changes. Refactoring, if applied correctly, is ideal for software evolution, since it is guaranteed that no new bugs are introduced.

This chapter presents and analyzes a number of refactorings. While the refactorings have been discusssed elsewhere, this work appears to be the first to discuss the analyses necessary to automate the refactorings.

Most refactorings discussed in this chapter are from the refactoring book by Martin Fowler [33]. Two additional simple low-level refactorings were noticed while we were examining an actual case of software evolution: Reverse Conditional and

Consolidate Duplicated Conditional Fragments. This chapter categorizes refactorings as low-level (Section 3.2) and high-level (Section 3.3).

Fowler provides the definition of each refactoring but does not discuss the complexities with respect to behavior preservation. His suggestion is to test the code and compare the output after each refactoring. This is an ad-hoc approach. Such an approach is not fool-proof because it relies solely on test cases. Moreover, it could put programmers in a situation where changes have been made but the refactored code produces different results. Testing code after refactoring is necessary but insufficient.

This chapter starts with seven properties defined by Opdyke that must be used to ensure behavior preservation. It describes characteristics and complexities of each refactoring. Refactoring complexities will be discussed based on properties identified by Opdyke (Section 3.1). An explanation of why and which analysis is required for each refactoring in order to keep semantics unchanged (Opdyke's 7th property) is also presented in this chapter. Most code examples are from real world projects. Some are from the Fluid Framework and some are CS552 students and instructors who generously donated their code.

3.1 Opdyke's Behavior Preservation Properties

Opdyke [73] has determined a set of properties of programs that must be checked to ensure behavior preservation. Such properties are:

1. Unique Superclass

2. Distinct Class Names

3. Distinct Member Names

4. Inherited Member Variables Not Redefined

5. Compatible Signatures in Member Function Redefinition

6. Type-Safe Assignments

7. Semantically Equivalent References and Operations

The first six properties are syntactic while the seventh property is semantic. The compiler can usually detect any violations of syntactic properties but not the semantic property. Checking syntax errors after refactoring is necessary but insufficient to guarantee behavior preservation. In order to ensure that a program after refactoring is semantically equivalent to the program before refactoring, the preconditions for each refactoring must be carefully defined.

One of the most well-known refactoring tools in the market is refactoring support in Eclipse IDE. Unfortunately Eclipse's refactoring support only checks the syntax of the code after applying refactoring. It does not check the conditions related to program semantics. It employs only the first six properties which do not fully ensure behavior preservation. Therefore, using Eclipse's refactoring tool is unsafe because it may change the semantics of the program. It is conceivable that Eclipse will integrate code smell detection with the refactoring tool in the future. In this case, this dissertation still has a positive contribution in the semantic analyses. Such

semantic conditions can be used in addition to the syntactic checks. Furthermore, the contributions from this work are also applicable to other object-oriented languages and can also be used as a guideline for those who want to implement code smell warnings for other types of refactorings.

3.2 Low-Level Refactorings

The definition of low-level refactoring in this work is somewhat different from that of Opdyke's. While Opdyke defines low-level refactorings as those that for which it is trivial to show that they are behavior preserving, this work considers a refactoring to be low-level if it does not involve complicated program analyses to ensure behavior preservation.

3.2.1 Rename

Renaming is the most frequently used refactoring. It usually takes place when we find that the name of a program entity does not represent its purpose. It is common that programmers do not the name right the first time.

Complexities

Renaming may sound simple but implementing such a refactoring while trying to preserve semantics is not trivial. Suppose a programmer wants to rename a method. Putting behavior preservation aside, one of the complexities involves updating all references to use the new name. With behavior preservation in mind, it is necessary

to check if the class already has a method with that name before renaming. If such a method exists, two things could happen, 1) if it has the same signature as the one to be renamed, a compile error occurs. 2) if they have different signatures, there may be no compile error but this renaming causes design change, because it introduces overloading which may cause overloading resolution to give a different result. More complexities arise since inheritance must also be taken into consideration. In the object-oriented programming world, looking at the class that defines the method alone is not sufficient. Though it may look like everything works the same way within one class, other classes in the same hierarchy may be affected by that change. We have to go up and down the inheritance hierarchy because renaming without such information can create overriding. Introducing an overloading/overriding method may change semantics of the program.

Figure 3.2 demonstrates careless method renaming. Though renaming `print` method to `display` in the class `FarewellMessage` does not introduce compile errors. Method `displayAndExtra` in figure 3.2b behaves differently. Unlike the former code, it shows message Hello instead of Good Bye.

Required Analysis

None.

```
private void updateWarehouse() {
  ...
  writeWarehouse();
}

private void exitWarehouse() {
  ...
  writeWarehouse();
  System.exit(0);
}

private void writeWarehouse() {
  ...
}
```

(a) before

```
private void updateWarehouse() {
  ...
  saveWarehouse();
}

private void exitWarehouse() {
  ...
  saveWarehouse();
  System.exit(0);
}

private void saveWarehouse() {
  ...
}
```

(b) after

Figure 3.1: Rename Method Refactoring

3.2.2 Extract Method

When a method is too long or is doing too much, we can extract a region of code and make a new method for it. Not only does extracting method make the code more readable, but it also promotes code reuse. An indirect result of extracting a method is improving the code maintainability. Sometimes a method is extracted when the programmer foresees substantial future changes such as adding more responsibilities to a method.

Complexities

Extract Method is done within a class so it is less complicated than refactorings that involve more than one classes. The newly extracted method must have access to all local variables it uses. Therefore, such variables must be passed into the extracted method as parameters. Fields are exceptions, since they are accessible through out the class. Figure 3.3 shows how a method is extracted. The programmer wants to extract

```
public class Message {
  void display() {
    System.out.println(‘‘Hello’’);
  }
}

public class FarewellMessage extends Message {
  void print() {
    System.out.println(‘‘Good Bye’’);
  }

  void displayAndExtra() {
    display();       //print Hello
    ...
    print();         //print Good Bye
  }
}
```

(a) before

```
public class Message {
  void display() {
    System.out.println(‘‘Hello’’);
  }
}

public class FarewellMessage extends Message {
  void display() {
    System.out.println(‘‘Good Bye’’);
  }

  void displayAndExtra() {
    display();       //print Good Bye
    ...
    display();       //print Good Bye
  }
}
```

(b) after

Figure 3.2: Careless Method Renaming

```
public void init() {
  JTextField textField = _itemPrice;
  int iCol = 5;
  textField.setColumns(iCol);
  textField.setFont(new Font(textField.getFont().getFontName(),
                        Font.BOLD,
                        textField.getFont().getSize()));
  textField.setBackground(pricePanel.getBackground());
  textField.setEditable(false);
  textField.setBorder(BorderFactory.createEmptyBorder());
  pricePanel.add(textField);
}
```

(a) before

```
public void init() {
  JTextField textField = _itemPrice;
  int iCol = 5;
  setLabel(pricePanel, textField, iCol);
  pricePanel.add(textField);
}

private void setLabel(JPanel pricePanel,
                        JTextField textField,
                        int iCol) {
  textField.setColumns(iCol);
  textField.setFont(new Font(textField.getFont().getFontName(),
                        Font.BOLD,
                        textField.getFont().getSize()));
  textField.setBackground(pricePanel.getBackground());
  textField.setEditable(false);
  textField.setBorder(BorderFactory.createEmptyBorder());
}
```

(b) after

Figure 3.3: Extract Method

a portion of code where several properties of textField are set. Since the extracted code refers to local variables: `pricePanel`, `textField`, `iCol`, they are added to the `setLabel` method signature.

More complexity arises if the extracted code contains an assignment to a local variable. If that local variable is not used after the extracted code, no other steps are required. If it is used, the extracted method must return the value of that local

```
static void shipment(Warehouse w, BufferedReader br) {
  System.out.println("Please enter order");
  Order o = new Order();
  o.read(br);
  Order l = o.ship(w);
  System.out.println("Back-ordered: ");
  l.write(System.out);
}
```

(a) before

```
static void shipment(Warehouse w, BufferedReader br) {
  Order o = readOrder(br);
  Order l = o.ship(w);
  System.out.println("Back-ordered: ");
  l.write(System.out);
}

static Order readOrder(BufferredReader br) {
  System.out.println("Please enter order");
  Order o = new Order();
  o.read(br);
  return o;
}
```

(b) after

Figure 3.4: Method Extraction that Requires a Return Statement

variable. This requirement is established to ensure that the original method still has access to the same object after the extraction. However, extracting code that has more than one assignment cannot be done because a method can only return one value. Consequently, it is necessary to check if there is any assignment to local variables in that region of code and if so, we have to further check the number of assignments. If there is only one assignment, we can proceed with the extraction process. If there is more than one assignment, the code cannot be extracted to a new method. Assignments to fields do not have this problem and do not require special treatments, since all methods have access to the class fields. Figure 3.4 illustrates the situation when a "return" statement has to be added to the extracted method.

```
int currentYear = 2012;
int totalCost = 0;
for (int i = 0; i < employeeList.size(); ++i) {
  Employee e = employeeList.get(i);
  int cost = e.getManMonth() * e.getStandardRate(currentYear);  //to be extracted
  totalCost = totalCost + cost;
}
```

(a) before

```
int currentYear = 2012;
int totalCost = 0;
for (int i = 0; i < employeeList.size(); ++i) {
  Employee e = employeeList.get(i);
  totalCost = totalCost + e.getManMonthCost(currentYear);
}

// A newly extracted method in Employee class
public int getManMonthCost(int year) {
  return e.getManMonth() * e.getStandardRate(year);
}
```

(b) after

Figure 3.5: Method Extraction inside a Loop

Required Analysis

Live Variable Analysis: Live variable analysis [69] is used to determine useless variables. It determines whether a variable will be used in the future. If the variable is not used, it is considered "dead" and can be removed. For method extraction, live variable analysis is used in a different aspect. We use the analysis to identify a set of variables that are used in the extracted code. The result of the analysis represents all live variables that must be passed as parameters into the newly extracted method.

Furthermore, we need live variable analysis to determine variable definition inside a loop. If a variable is redefined inside the loop, the extracted method must return the value for that variable as demonstrated in Figure 3.5

```
if (!isStronger(a, b)) {
  winner = b;
} else {
  winner = a;
}
```

(a) before

```
if (isStronger(a, b)) {
  winner = a;
} else {
  winner = b;
}
```

(b) after

Figure 3.6: Reverse Conditional

3.2.3 Reverse Conditional

Reverse conditional refactoring is one of the refactorings that we identified during the early stages of this research. At that time, we looked at the history of two classes written in Java, studied changes made in 35 versions over 3 years of development. We then determined which changes could be done using refactoring. The code is written and modified by three different people and the final length of the code is XXX lines so it is a non-trivial project.

It is noteworthy to remark that the name Reverse Conditional is defined by Bill Murphy and Martin Fowler. Not long after we discovered this refactoring, they found such a refactoring and gave it an official name on www.refactoring.com.

Reverse Conditional is a very simple refactoring since it does not require any analysis. Proving that this refactoring preserves program behavior is trivial. It is simple to perform and is useful, since it can improve the readability of the code. Both Figure 3.6(a) and Figure 3.6(b) give the same results, the latter code makes more sense than the former code.

Complexities

None.

Required Analysis

None.

3.2.4 Consolidate Duplicated Conditional Fragments

This is another refactoring we discovered that is not in Fowler's book. It is recommended when similar fragments of code exist in every branch of a conditional statement. It is an instance of *code hoisting*. Code hoisting is an optimization to reduce code size by eliminating statements that occur in multiple code paths from a single common point in the CFG. Though performing code hoisting on conditional statements does not improve code speed as it does for loop invariants, it simplifies the code and makes the code more manageable. Figure 3.7 shows how statements are consolidated.

Complexities

In order to safely move the duplicated code fragment out of a conditional statement, we must be certain that such fragment is not affected by the code nor have effects on the code inside the branches. In other words, the code inside the branches and the duplicated code fragment must be independent of each other. The code in Figure 3.7 requires no analysis and can be moved out because the statement is at the end of each branch. Hence, there is no statement to check for conflicts in effects. If the statement to be moved is at other locations, it is a general case of statement reordering which requires Effects Analysis. The complexity of statement reordering will be further

```
depth = 0;
if (first < second) {
  ...
  depth = depth + 1;
} else {
  ...
  depth = depth + 1;
}
```

(a) before

```
depth = 0;
if (first < second) {
  ...
} else {
  ...
}
 depth = depth + 1;
```

(b) after

Figure 3.7: Consolidate Duplicate Conditional Fragments

described in Section 3.2.5.

Required Analysis

Effects Analysis.

3.2.5 Swap Statements

Swapping statements is primarily changing the order of statements. Figure 3.8 demonstrates a careless swap. Swapping two statements inside the while loop without checking dependencies causes method `factorial` to act differently. Using the code in our example to compute `factorial(3)`, Figure 3.8(a) returns 6 whereas Figure 3.8(b) returns 2.

```
// factorial(3) = 6
int factorial(int x) {
  int fact = 1;
  int y = x;
  while (y > 0) {
    fact = fact * y;
    y = y - 1;
  }
  return fact;
}
```

(a) before

```
// factorial(3) = 2
 int factorial(int x) {
  int fact = 1;
  int y = x;
  while (y > 0) {
    y = y - 1;
    fact = fact * y;
  }
  return fact;
}
```

(b) after

Figure 3.8: Bad Statement Swap

Complexities

Swapping two statements sounds straightforward but when semantic preservation must be taken into account, it is no longer trivial. Analysis is needed because two statements can be swapped only if they do not "interfere" with each other.

Behavior preservation cannot be guaranteed if a statement does not terminate or throws an exception. The semantic conditions in this research are based on an assumption that both statements terminate normally in all cases.

Required Analysis

Effects Analysis: Boyland and Greenhouse introduce an effects system for object-oriented programming [39]. They define two kinds of effects: *read effects* and *write effects*. Read effects are those that may read the contents of a mutable state. Write effects are those that may change or read the contents of a mutable state. Statements interfere with each other if they are flow dependent (write, read), output dependent (write, write) or anti-dependent (read, write). In other words, two effects conflict if at least one is a write effect and they involve *targets* that may overlap. A target is a mutable variable or a mutable state of an instance of a class.

Let s_1 and s_2 be two consecutive statements, x be an arbitrary variable or a mutable object. Statements s_1 and s_2 do not interfere with each other if:

$$\forall x : s_1 \text{ and } s_2 \text{ that have an overlapped target } x$$
$$\mathsf{Effect}(s_1) \not\ni (\mathsf{write}, x) \wedge \mathsf{Effect}(s_2) \not\ni (\mathsf{write}, x)$$

Let's revisit Figure 3.8 and see how effects analysis plays a crucial role in this refactoring. According to the condition discussed previously, if statements `fact = fact - 1` and `y = y - 1` have effects on a target variable, both statements must not write it. However, when we compute the read/write effects of the two statements, we get the following results:

$$[5] : \texttt{fact} = \texttt{fact} * \texttt{y}$$
$$\mathsf{Effect}([5]) = \{(\mathsf{read}, \texttt{fact}), \underline{(\mathsf{read}, \texttt{y})}, (\mathsf{write}, \texttt{fact})\}$$

$$[6] : \mathsf{y} = \mathsf{y} - 1$$

$$\mathsf{Effect}([6]) = \{(\mathsf{read}, \mathsf{y}), \underline{(\mathsf{write}, \mathsf{y})}\}$$

As seen above, statement [6] writes y while statement [5] reads it which violates the semantic preservation condition. Hence, these two statements cannot be swapped without affecting program behavior.

Our condition is not too restrictive. Though the condition requires two statements to be consecutive, it is generalized and can be applied to non-consecutive statements. Swapping non-consecutive statements is literally performing a series of swaps on two consecutive statements. Suppose there is a series of statements (s_1; s_2; $\ldots s_i$;) and we want to swap statements s_i with s_1. There will be $2i - 3$ swaps *i.e.,* $i - 1$ swaps from moving s_i in front of s_1 and $i - 2$ swaps from moving s_1 after s_{i-1}. If the statement is being moved downward in the CFG, we have to compute the effects of such a statement and those below it. Similarly, if it is being moved up the CFG, we have to check its effects against the effects of all statements above it. Below exhibits steps taken when swapping s_i with s_1.

$$
\begin{array}{llllllllll}
\underline{s_1}; & s_2; & \ldots & s_{i-1}; & \underline{s_i}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
\underline{s_1}; & s_2; & \ldots & \underline{s_i}; & s_{i-1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
 & & & & \vdots & & & & \\
\underline{s_1}; & \underline{s_i}; & s_2; & \ldots & s_{i-1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
\underline{s_i}; & \underline{s_1}; & s_2; & \ldots & s_{i-1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
\underline{s_i}; & s_2; & \underline{s_1}; & \ldots & s_{i-1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
 & & & & \vdots & & & & \\
\underline{s_i}; & s_2; & \ldots & \underline{s_1}; & s_{i-1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n; \\
\underline{s_i}; & s_2; & \ldots & s_{i-1}; & \underline{s_1}; & s_{i+1}; & \ldots & s_{n-1}; & s_n;
\end{array}
$$

It is worth noting that the implementation does not have to perform the intermediate swaps. It only has to check the effects $2i-3$ times. After all effects are checked, we can change the order of s_1 and s_i in one swap.

Special case: If the statement is inside a loop (for, while) and we want to move it up (outside), we must check all effects against itself and their conditions. Moreover, Reaching Definition Analysis [69] is needed to determine *loop invariant*. However, if a statement is being moved outside an if statement, an extra check is needed *i.e.,* it must occur in every branch. See Consolidate Conditional Fragment in Section 3.2.4.

3.3 High-Level Refactorings

High-level refactorings are those that are difficult to implement and require program analysis to ensure behavior preservation. Since the definition of high-level refactoring in this work is different than Opdyke's, many may refer some refactorings in this section as low-level.

3.3.1 Inline Method

Method inlining is basically an inverse of method extraction. This particular refactoring inlines all invocations of a method and remove its declaration. Sometimes a programmer considers inlining a method if he finds that such a method does not have a lot of responsibilities. There are no apparent rules as to when to extract or inline a method. The decision is made based on each individual's preference.

Complexities

Inline method involves an opposite issue of that of Extract Method. It is normal that a method to be inlined has a number of local variable definitions. When inlining, such variables will be introduced to the target method. Hence, our behavior preserving task involves checking if the target has already defined variables with the same names. If the variable names conflict, they have to be renamed. Otherwise, it will introduce variable re-definitions and cause compile errors (if in the same scope) or cause the program to behave differently (if in a different scope).

Though method inlining involves changing the code in a single class, it could affect

other parts of the program depending on the type of the methods. No further checks are required for `private` method because it is invisible to other classes. Hence, the changes have no effects on them. The same issue applies to `final` methods. When a method is declared `final`, no other classes can override it so it is safe to perform inlining.

However, if a method could be overridden, there is a high risk in changing the behavior. If the subclasses override the method that is being inlined, their behavior will definitely be affected. Consider Figure 3.9 which shows an instance of the problems. Suppose the programmer wants to inline method `restockItem` in class `Merchant`. It is obvious that there is no behavioral change for class `Merchant`. However, `Retailer`, a subclass of `Merchant` redefines (overrides) `restockItem` method. Inlining `restockItem` into `checkStock` affects `Retailer`'s behavior. `Retailer.checkStock` puts 5 items on the shelves before inlining but it puts 20 items after the change.

Required Analysis

None, if the method to be inlined is `final` or `private` *i.e.,* the method cannot be overridden. If it is neither `final` nor `private`, an approach similar to *devirtualization* is needed. Devirtualization is an optimization technique to reduce the overhead of virtual (dynamic) method call by replacing a virtual method call with a particular method of a class. Though Inline Method refactoring does not focus on optimization, their mechanics and complexities are the same.

Class Hierarchy Analysis (CHA) [17] is one of the most well-known techniques for

```
// Merchant.checkStock - restock 20 items if no item is in stock
class Merchant {
  void checkStock(Item item) {
    if (getQuantity(item) == 0)
      restockItem(Item item);
  }

  void restockItem(Item item) {
    _shelves.put(item, 20);
  }
}

// Retailer.checkStock - restock 5 items if no item is in stock
class Retailer extends Merchant {
  void restockItem(Item item) {
    _shelves.put(item, 5);
  }
}
```

(a) before

```
// Merchant.checkStock - restock 20 items

class Merchant {
  void checkStock(Item item) {
    if (getQuantity(item) == 0)
      _shelves.put(item, 20);
  }
}

// Retailer.checkStock - now restocks 20 item!!

class Retailer extends Merchant {
  void restockItem(Item item) {
    _shelves.put(item, 5);
  }
}
```

(b) after

Figure 3.9: Careless Inline Method

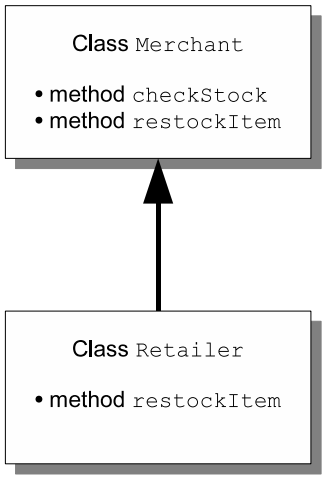


Figure 3.10: Class Hiearchy Analysis

devirtualization. It is a static analysis that determines a set of possible targets of a dynamic method call with the class hierarchy of the whole program. If it is possible to determine that there is no overridden method, the dynamic method call can be replaced with inlined code. CHA can be performed without programmer intervention. Ishizaki *et al.* claim the accuracy of class hierarchy analysis can be improved by adding type analysis and preexistence analysis [45].

Figure 3.10 illustrates a result of class hierarchy analysis for `Merchant` and `Retailer` code. Let's consider the situation where a programmer wants to inline method `restockItem` into `checkStock`. Using CHA, we are able to determine that class `Retailer` overrides method `checkStock` which reveals that the inlining should not take place.

A thorough discussion on method inlining can be found in the work by Detlefs and Agesen [20]. Sometimes the system's performance gets worse because type casts

are usually inserted to preserve typability. Glew and Palsberg [35] discuss type-safe inlining. There are many techniques for devirtualization not discussed here.

3.3.2 Move Method

Move Method is used to move a method to a different class than the one it is defined in. A method is usually moved when the programmer finds that such a method does not belong in the class in which it is defined. It is collaborating too much with other classes. Such collaboration includes using or being used by more features of other classes than the class it lives in. Consequently, classes become too highly coupled which is poor design.

Moving a method to the superclass/subclass is also known as Pull Up Method/ Push Down Method. An example of a method move between unrelated classes is depicted in Figure 3.11.

Complexities

One of the issues when moving a method is to avoid name conflict. Similar to renaming, it is necessary to check that the target class does not have a method with the same name. Checking for name conflicts must also be done on the entire inheritance hierarchy to ensure behavior preservation. The reasons behind this issue are already discussed in Section 3.2.1.

The other issue involves the possibility that the method being moved overrides that of its superclass. If it is overridden, any subclasses that override the method will

Refactoring	Analysis
Rename	None
Move Method	None
Extract Method	Live Variable Analysis
Inline Method	Class Hierarchy Analysis
Reverse Conditional	None
Consolidate Duplicated Conditional Fragments	Effects Analysis
Swap Statements	Effects Analysis
Move Field to Component Class	Uniqueness Analysis
Move Method to Component Class	Uniqueness Analysis
Move Field to Aggregate Class	Uniqueness Analysis

Table 3.1: Refactorings and Analyses

be disturbed. A more detailed discussion was provided in Section 3.3.1. To ensure behavior preservation, our work will not allow the move if it is an overriding method.

After a method is moved, it must still be accessible from its callers. To avoid any compile errors, the modifier of the moved method must be changed accordingly. For simplicity, this work always makes the moved method `public`.

Maruyama and Takayuki also describe the complexities and security issues concerning Pull Up Method and Push Down Method [61]. Their concern is mainly on preventing unauthorized external code to access sensitive data. The degree of confidentiality or access level are measured and if the access level of the modified source code is decreased or downgraded, they consider the modified code to be vulnerable to attackers.

Required Analysis

None.

```
public class Driver {
  ...
  private static void writeWarehouse(String strFileName) throws IOException {
    File objFile = new File(strFileName);
    Warehouse objWh = Warehouse.getInstance();

    if (objFile.exists() && objFile.canWrite()) {
        PrintStream psWrite = new PrintStream(new FileOutputStream(objFile));
        objWh.write(psWrite);
    } else {
        objWh.write(System.out);
    }
  }
}
```

(a) before

```
public class Driver {
  ...
  public void init() {
    ...
    Warehouse.writeWarehouse(``warehouse.dat'');
  }
}

public class Warehouse {
  public static void writeWarehouse(String strFileName) throws IOException {
    File objFile = new File(strFileName);

    if (objFile.exists() && objFile.canWrite()) {
      PrintStream psWrite = new PrintStream(new FileOutputStream(objFile));
      this.write(psWrite);
    } else {
      this.write(System.out);
    }
  }
}
```

(b) after

Figure 3.11: Move Method

3.3.3 Convert Inheritance into Aggregation

Inheritance represents "is-a" relationship while aggregation/components represents "has-a" or "whole/part" relationship. According to Foote and Opdyke [32], Convert Inheritance into Aggregation refactoring supports software evolution and reuse. Changing inheritance into aggregation offers many benefits. Such benefits include "part" encapsulation where Gamma *et al.* call it "black-box reuse" [34]. The term black-box is derived from the fact that the internal structure of objects is invisible to the clients. The implementation of a subclass may become so dependent with its superclass that changes in the superclass will cause the subclass to change. With aggregation, there are fewer implementation dependencies. Moreover, an aggregate could have more than one instance of a component class. While inherited parts are static, aggregation allows a class to change their components at runtime.

Each programmer has different viewpoints and they do not always choose the correct mechanism for modeling a particular relationship [73, 46]. However, as a program evolves, the relationships between classes become more evident. Convert Inheritance into Aggregation refactoring allows the programmer to convert a subclass/superclass relationship into an aggregation. Every behavior inherited by the subclass from its superclass before this refactoring will be delegated to a new component, which is an instance of the old superclass.

Consider a class `TwoDimensionalArray` that defines a field `elements` and methods to access and manage its elements and a class `Matrix` that implements matrix operations. The original design is that the `Matrix` class inherits the `TwoDimensionalArray`

class. It is later realized that not all matrices should be conveyed in two-dimensional array, for instance using other kinds of representation for a sparse matrix would be more efficient. A matrix is not a representation. It "has" a representation. Hence, it is advisable to convert the original design from inheritance to aggregation.

Complexities

Converting inheritance into aggregation is complicated because it involves changing an inheritance hierarchy. The first thing to consider when inheritance changes is the accessibility to inherited methods and fields as they must still be accessible from the aggregate class when the changes occur. Method and field references must be updated not only in the aggregate class, but also in its clients.

Updating references to inherited fields and methods can be tricky. There is a possibility that the aggregate class has a subclass that overrides a method of the component. Consider a situation where a class `UpperTriangularMatrix` which is a subclass of `Matrix` overrides the method `putElement` so that it prevents putting values other than 0 below the diagonal. The scope of reference updates must be expanded to cover all subclasses of the aggregate. Opdyke does not mention this issue in his work.

Required Analysis

None.

```
public class Automobile {
  Engine autoEngine;
  int numOfPassenger;
  Tire leftFrontTire;
  Tire rightFrontTire;
  Tire leftRearTire;
  Tire rightRearTire;
  ...
}
```

Figure 3.12: Aggregate/Component Relationship

3.3.4 Move Members between Aggregate and Component Class

As discussed in the previous section, aggregation/components represents "has-a" or "whole-part" relationship. The idea of the Whole-Part pattern is to introduce a component (the whole) that encapsulates smaller objects (the parts) which prevents clients from accessing these parts directly.

For instance, an automobile is composed of a body, four tires, a steering wheel and an engine. The car itself is an aggregate component while the tires, steering wheel and engine are parts (components) of the car. Figure 3.12 shows aggregate and component classes. Automobile is an aggregate class, while Engine and Tire are component classes.

It is sometimes necessary to move class members between aggregate and component classes. The original design may be improper or unsuitable with the current requirements. It may grant or restrict too much client's access. Moving members to a component class restricts access from other clients. On the contrary, moving members to an aggregate class removes permits direct access from clients.

Moving members between aggregate and component classes is not easy to accomplish without error. It is a special case of Move Method so it inherits all complexities that were discussed in Section 3.3.2. Typically, the part can only belong to one whole at a time (Figure 3.13). In addition, each part in one whole must be unique (Figure 3.14). Checking these conditions requires uniqueness analysis. Performing these changes automatically with precondition checks eliminates the possibility of inadvertently changing behavior.

- Move Field to Component requires uniqueness analysis on the component field, not the field being moved before moving members from an aggregate to a component class. It is required to determine whether a variable qualifies as a component member variable *i.e.,* every object assigned to it is not also assigned to another component variable.

- Move Method to Component does not require a component to be an exclusive component of an aggregate class. Methods are different from fields as they are considered as services. However, it does require that a reference to the component class must be reachable whenever a method is called. In addition, the method must still be able to refer to an instance of the aggregate after the move. They are the same requirements as Move Method discussed in Section 3.3.2. The reference in the latter case could be carried out by adding an extra parameter to the method which creates delegation.

- Move Field into Aggregate is an inverse operation of Move Field to

```
public class AutomobileFactory {
  Automobile car = new Automobile();
  Tire temp = new Tire();
  car.leftFrontTire = temp;
  car.rightFrontTire = temp;
}
```

Figure 3.13: Nonexclusive Components (1)

```
public class AutomobileFactory {
  Automobile car1 = new Automobile();
  Automobile car2 = new Automobile();
  Tire temp = new Tire();
  car1.leftFrontTire = temp;
  car2.leftFrontTire = temp;
}
```

Figure 3.14: Nonexclusive Components (2)

COMPONENT which also requires uniqueness analysis. For the same reason, variables from the component class can be moved to an aggregate class only if the component is exclusive. If an aggregate has more than one instance of a component, moving a field to an aggregate will require adding a variable for each instance.

Complexities

Moving members between aggregate and component classes is a variant of MOVE METHOD and MOVE FIELD. In addition to complexities discussed in Section 3.3.2, it requires further analysis. It is more complicated than a normal move because it is required that such a component be exclusive. If a component is nonexclusive, the member cannot be moved because doing so will change program behavior.

Consider the code shown in Figure 3.15(a), the Automobile class contains 4 tires and one warranty expiration for each tire. For design reasons, the programmer wants

```
public class Automobile {
  Engine autoEngine;
  int numOfPassenger;
  Tire leftFrontTire;
  Tire rightFrontTire;
  Tire leftRearTire;
  Tire rightRearTire;
  WarrantyInfo warrantyExpirationLeftFrontTire;
  WarrantyInfo warrantyExpirationRightFrontTire;
  WarrantyInfo warrantyExpirationLeftRearTire;
  WarrantyInfo warrantyExpirationRightRearTire;
  ...
}

public class Tire {
  ...
}
```

(a) before

```
public class Automobile {
  Engine autoEngine;
  int numOfPassenger;
  Tire leftFrontTire;
  Tire rightFrontTire;
  Tire leftRearTire;
  Tire rightRearTire
  ...
}

public class Tire {
  ...
  WarrantyInfo warrantyExpiration;
}
```

(b) after

Figure 3.15: Moving members

to move warranty expiration to the Tire class. Since Tire is a component class of Automobile class, we have to check if all instances of `rightFrontTire`, `leftFrontTire`, `rightRearTire` and `leftRearTire` are unique. If they pass uniqueness analysis, warranty information can be moved to the Tire class safely as seen in Figure 3.15(b). Figure 3.13 and Figure 3.14 illustrate two scenarios of nonexclusive components. In Figure 3.13, the same tire is assigned as the left front tire and right front tire. In this case, both `leftFrontTire` and `rightFrontTire` are not exclusive components. Let's look at Figure 3.14. This time the same tire is assigned as the left front tire of two different cars. `car1.leftFrontTire` and `car2.leftFrontTire` are not unique. Therefore, `leftFrontTire` of class `Automobile` cannot be designated as an exclusive component.

Required Analysis

Uniqueness analysis is an analysis that determines whether a variable or an object is unique at a specific program point [5, 11].

3.3.5 Create Abstract Superclass

Using an abstract superclass is a classic design pattern [34]. An abstract superclass is desirable when two sibling classes implement and use common features. An abstract superclass supports code reuse and indirectly reduces a number of duplicated code.

Opdyke and Johnson [74] have shown that it is feasible to create abstract superclass by a series of atomic refactorings: Move Method and Move Field which

```
public class Car {
  int mpg;
  double fuelInTank;

  public Car() {
    mpg = 35;
    fuelInTank = 6.056;
  }
  public void pumpGas(double g) {
    fuelInTank = fuelInTank + g;
  }
  public void drive(double miles) {
    double used = miles / mpg;
    fuelInTank = fuelInTank - used;
  }
}

public class Truck {
  int mpg;
  double fuelInTank;

  public Truck() {
    mpg = 25;
    fuelInTank = 11.355;
  }
  public void pumpGas(double g) {
    fuelInTank = fuelInTank + g;
  }
  public void drive(double miles) {
    double used = miles / mpg;
    fuelInTank = fuelInTank - used;
  }
}
```

(a) before

```
abstract class Automobile {
  int mpg;
  double fuelInTank;

  public void pumpGas(double g) {
    fuelInTank = fuelInTank + g;
  }

  public void drive(double miles) {
    double used = miles / mpg;
    fuelInTank = fuelInTank - used;
  }
}

class Car extends Automobile {
  public Car() {
    mpg = 35;
    fuelInTank = 6.056;
  }
}

class Truck extends Automobile {
  public Truck() {
    mpg = 25;
    fuelInTank = 11.355;
  }
}
```

(b) after

Figure 3.16: Create Abstract Superclass

could be done after a new class with a unique name is created.

Complexities

One of the complexities of Create Abstract Superclass is finding common code that can be migrated to the abstract superclass.

Another complexity relates to how we deal with common abstractions. Since it

involves moving common methods to the abstract superclass, it inherits some complexities of Move Method which have been discussed in Section 3.3.2. However, some of the condition checks could be skipped because the newly created abstract class is empty and has no members defined. There are no name conflicts in fields and methods. In addition to Move Method, fields that are referenced by the common code must also be moved to the superclass.

Figure 3.16(a) shows that class `Car` and `Truck` implement similar sets of methods. This situation is an instance of duplicated code. Introducing an abstract class `Automobile` which centralizes the responsibilities obviously yields a better design. The changes that had to be done in two places formerly, can be taken care of in just one place.

Required Analysis

Duplicated Code Analysis. Furthermore, they must already have a common superclass or no other superclass than `java.lang.Object`.

3.4 Discussion and Summary

In this chapter, we provide the descriptions and complexities of each refactoring as well as the analyses that it requires to ensure behavior preservation. For some refactorings, the behavior preservation condition check is more complicated than applying

refactoring itself to the code. Semantic properties for some refactorings require extensive program analysis. For instance, Swap Statements and Consolidate Conditional Fragments require effects analysis. Moving fields between aggregate and component requires uniqueness analysis.

In general, the following conditions must be considered when refactoring.

Redefinitions When a new entity is created, it is necessary to determine if the name conflicts with the existing declarations to avoid redefinitions.

- local variables: could cause compile errors and semantic change depending of the scope.
- fields: could cause compile errors and hiding
- methods: could introduce overriding and overloading.

Effects of the changes We need to take into consideration how the changes affect other parts of the program. In object-oriented programming, it is inadequate to analyze only the class that is being changed because it could be inherited. For instance, inlining a method that could be overridden requires some devirtualization analysis such as class hierarchy analysis.

For many refactorings, the complexities come from ensuring *reachability* and updating *references*.

3.5 Summary

This chapter has shown that a simple refactoring could change the program semantics. Program behavior is sensitive to changes even small ones. Every change made to the program should not be taken for granted.

Chapter 4

Code Smells

Code smells are design flaws that can be solved by refactorings. They are considered as flags to the developer that some parts of the design may be inappropriate and that it can be improved. For the purpose of this work, we discuss a few representative code smells. There are a lot of code smells not mentioned nor developed in this work. A thorough catalog of code smells can be found in Fowler's refactoring book [33]. As this work focuses on program analysis, code smells discussed in this work include those that require analyses. Though this work develops only a subset of the code smells, it provides some grounds which can be adapted to other types of code smells.

4.1 Duplicated Code

Duplicated code (code clone) is one of the most common problems in software development. Previous work [55, 4] suggests that about 5-10% of the source of large

scale programs is duplicated code. A number of recent studies show that many well-known open source systems have substantial duplicated code problems. The Java JDK (2002) is 21-29% duplicated [48]. The Linux kernel (2002) is estimated at 15-25% duplicated [1]. The GNU compiler (1999) is about 19% duplicated [22].

Duplicated code is usually caused by copy-and-paste action with an intent to reuse the code. This technique is easy and cheap during software development but it is considered bad practice. It makes software maintenance more complicated in many ways: 1) If there exist bugs or errors in the original code, they will be propagated with every duplication. 2) More generally, when an instance of duplicated code needs to be changed, all other duplicated instances must also be modified. 3) Furthermore, duplicated code makes performing *code auditing* more difficult. Code auditing is an analysis of source code with attempts to reduce software vulnerabilities.

4.1.1 Detecting Duplicated Code

Researchers have developed a number of approaches for so called "clone detection" [8, 22, 52, 48]. These approaches are:

- **text-based** [22]. This approach is language independent but it can only detect exact textual matches. Similar code with different variable names is considered different.

- **graph-based** [52, 54]. This approach uses program dependence graphs (PDGs). Krinke [54] finds similar code based on identifying maximal similar subgraphs.

Komondoor and Horwitz [52] use program slicing which allows non-contiguous clones to be detected.

- **token-based** [48]. Token-based approach lies somewhere in between the text-based and syntax-based. The source code are tokenized using a lexer. It is able to detect non-exact matches. It is one of the most effective aproach that balances soundness and speed.

- **syntax-based** [8]. Baxter and others use abstract syntax trees to detect exact match or near-miss clones. This approach can be applied to arbitrary program fragments. However, it cannot detect a clone where statements are arranged in different order.

Each of the discussed approaches has its pros and cons. This work chooses to use the syntax-based method because it is simple to implement in our AST-based analysis framework.

4.1.2 Refactorings for Duplicated Code

Duplicated code can be removed by different types of refactorings depending on where duplicates are found.

1. If they are in different methods of the same class, duplicated code can be removed by Extract Method.

2. If they are in two sibling classes, use Extract Method and/or Pull Up Method.

```
private void updateItemPanel() {
  Item item = getItem();
  int q = getQuantity();
  if (item == null) {
    _itemPanel.clear();
  } else {
    _itemPanel.setItem(item);
    int inStock = Warehouse.getInstance().getQuantity(item);
    _itemPanel.setInstock(q <= inStock && 0 < inStock);
  }
}
```

Figure 4.1: Feature envy

3. If they are in two unrelated classes, use Extract Class and/or Extract Method.

4.2 Feature Envy

Feature Envy occurs when a method seems to be more interested in some other class than the one it is defined in. It designates improper coupling between classes. An instance of Feature Envy is shown in Figure 4.1. Method `updateItemPanel` is defined in class `OrderItemPanel`. However, it is mostly interested in `ItemPanel`, since it invokes ItemPanel's methods on the field `_itemPanel` several times. Such invocations could be moved to the `ItemPanel` class in order to centralizes manipulations to the defining class.

4.2.1 Detecting Feature Envy

Feature Envy is a sign of improper coupling and cohesion. With this knowledge, cohesion and coupling measures seem to be the best candidates in finding feature envy in a class. We have done experiments on a number of cohesion and coupling metrics.

Unfortunately, none of them performed well when it comes to detecting feature envy. The results from cohesion and coupling measures are too broad as they can only tell which class is not cohesive and/or highly coupled with other classes. They do not tell which method has feature envy. Therefore, we developed a new metric to detect feature envy. The new metric will be discussed in more details in chapter 5.

4.2.2 Refactorings for Feature Envy

Move Method is required for all instances of feature envy. The purpose of this refactoring is to put a method in a class that is more suitable. The pre-condition for Move Method and its complexities are discussed in section 3.3.2. In addition to Move Method, Extract Method is needed if the problematic code does not cover the entire method. It is used as an intermediate step before the actual move.

Let's revisit an example in Figure 4.1. Method `updateItemPanel` has a portion of code that seems to be more interested in ItemPanel. It calls `clear`, `setItem` and `setInstock` on the field `_itemPanel`. Since those calls are inside an if statement, the if that covers method invocations on `_itemPanel` could be extracted as shown in Figure 4.2(a) and moved to `ItemPanel` class as showned in Figure 4.2(b). In this particular example, `ItemPanel` is a component of `OrderItemPanel`. Hence, a special case of Move Method that is Move Method to Component is used.

```
private void updateItemPanel() {
  Item item = getItem();
  int q = getQuantity();
  doUpdate(item, q);
}

private void doUpdate(Item item, int quantity) {
  if (item == null) {
    _itemPanel.clear();
  } else {
    _itemPanel.setItem(item);
    int inStock = Warehouse.getInstance().getQuantity(item);
    _itemPanel.setInstock(q <= inStock && 0 < inStock);
  }
}
```

(a) Extract Method `doUpdate`

```
public class OrderItemPanel {
  ...
  private void updateItemPanel() {
    Item item = getItem();
    int q = getQuantity();
    _itemPanel.doUpdate(item, q);
  }
}

public class ItemPanel {
  public void doUpdate(Item item, int quantity) {
    if (item == null) {
      clear();
    } else {
      setItem(item);
      int inStock = Warehouse.getInstance().getQuantity(item);
      setInstock(q <= inStock && 0 < inStock);
    }
  }
}
```

(b) Move Method `doUpdate` to Component `ItemPanel`

Figure 4.2: Remove Feature Envy by Refactorings

4.3 Data Class

A class that contains nothing but fields and get/set methods is called *data class*. When a class has no responsibility other than handing its data to the outsiders, it implies that its data is being manipulated by other classes. In practice, data should be encapsulated and not be exposed to others. Consider Figure 4.3(a), class `ItemList` only defines a constructor and a `getItemList` method which their only references are in `OrderItemWindow` class. `OrderItemWindow` solely controls `_itemList` after creating an instance and obtaining the data (`objItemList.getItemList`). The whole class can be moved to `OrderItemWindow` and deleted as seen in Figure 4.3(b).

4.3.1 Detecting Data Class

Figure 4.5 describes an algorithm that this research uses to detect a data class. Each class is evaluated by its members. If a class only has fields and "special" methods, it is considered to be a data class. Special methods include constructors and set/get methods. The tree structures of set/get methods are established as a benchmark in Figure 4.4. A method is determined by comparing its AST with the pre-defined patterns. If they match, it is marked as a special method. Fields and constructors do not need special treatments because they can be detected firsthanded by their node types.

```
abstract public class OrderItemWindow {
  ...
  protected static JList _itemList;

  public OrderItemWindow(JFrame objParentFrame,
                         Order objOrder,
                         OrderItem objOrderItem) {
    ...
    ItemList objItemList = new ItemList();
    _itemList = objItemList.getItemList();
  }
}

public class ItemList extends JList {
  private JList _itemList;
  DefaultListModel _listModel = new DefaultListModel();

  public ItemList() {
    Iterator iter = Catalog.getInstance().getIterator();
    while(iter.hasNext())
      _listModel.addElement((Item)iter.next());
      _itemList = new JList(_listModel);
  }

  public JList getItemList() {
    return _itemList;
  }
}
```

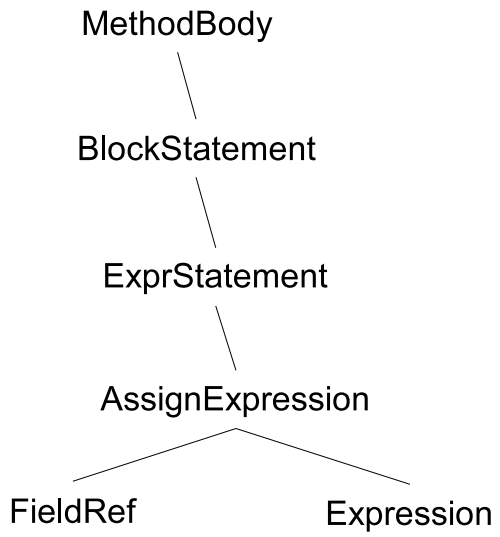
(a) before

```
abstract public class OrderItemWindow {
  ...
  protected static JList _itemList;

  public OrderItemWindow(JFrame objParentFrame,
                         Order objOrder,
                         OrderItem objOrderItem) {
    ...
    Iterator iter = Catalog.getInstance().getIterator();
    DefaultListModel _listModel = new DefaultListModel();
    while (iter.hasNext())
      _listModel.addElement((Item) iter.next());
    _itemList = new JList(_listModel);
  }
}
```

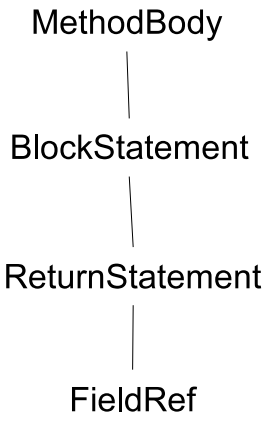
(b) after

Figure 4.3: Data Class




```
{  _field = ... ;  }
```

(a) set method




```
{
  return  _field = ... ;
}
```

(b) get method

Figure 4.4: Special Method Patterns

```
methods = {a method defined in a class}
for each method ∈ methods do {
    if method's structure matches special method's patterns
      return false
      }
      return true
```

Figure 4.5: Algorithm to detect data class

4.3.2 Refactorings for Data Class

There are two options to handle a data class. We could delete it after moving its class members to other class or assign more responsibilities to it.

Delete Data Class - For all fields and methods declared in the data class, we apply Move Field and Move Method to relocate the data class's members to other classes that uses them. Then, the data class is deleted after the move is complete. However, the target classes could be in the same inheritance hierarchy. Adding features to every class will introduce field redefinitions and method overriding. To conservatively prevent any syntactic and semantic problems, features should be added to only one class. Determining which class in the hierarchy is the most suitable can be very complicated.

Keep Data Class - Adding more responsibilities to the data class is one other alternative. Move Method that uses members of the data class from other classes.

Nonetheless, a data class could be a growing class whose features have yet to be implemented. The tool implemented in this research does not take that into

consideration. It is left up to the developer to make the final decision whether the suspected class is indeed a data class.

4.4 Switch Statement

A switch statement is basically another syntactic form of if-else statement. It makes the code more readable. In imperative programming, programmers usually use a switch statement to make polymorphic calls. However, it is different in object-oriented programming as polymorphism is handled automatically by the compiler also using *dynamic dispatch*. In object-oriented programming, a switch statement should not be used in place of polymorphism.

It is believed that replacing conditionals with polymorphism could cause the performance to degrade. Polymorphism resolution introduces extra processor time to consult the method lookup table (Java) or virtual function table (C++). Therefore, programmers usually argue that the cost of refactoring is too high. Demeyer determines the performance cost in replacing a conditional with polymorphism [18]. According to his experiments on C++ programs using the best possible optimization, virtual functions and switch statements have similar overhead. He also concludes that refactorings that move behavior close to data improve maintainability without sacrificing performance.

```
class Student {
  public static int UNDERGRAD = 1;
  public static int GRADUATE = 2;
  public static int DISSERTATOR = 3;
  static double segregated_fee = 300;
  private int num_credits;
  private int classification;

  public Student(int c, int cls) {
    num_credits = c;
    classification = cls;
  }

  public double getTuition() {
    switch (getClassification()) {
      case UNDERGRAD:
        return segregated_fee + 300 * num_credits;
      case GRADUATE:
        return segregated_fee + 350 * num_credits;
      case DISSERTATOR:
        return segregated_fee + 200 * num_credits;
    }
  }

  public int getClassification() {
    return classification;
  }
}
```

Figure 4.6: Poor Use of Switch Statement

4.4.1 Detecting a Switch Statement

For most code smells, the most difficult part is determining how to locate them and the easier part is the removal. Unlike other code smells, removing a switch statement is not as easy as locating it. Finding a switch statement in general is not difficult. We can perform a tree walk on the AST. Then, each switch statement found is added to a set of smells. However, reporting all switch statements found in the code is not desirable. There will be a lot of *false positives*. A false positive, in this case, is a switch statement that is not used to represent polymorphism. For instance, using a switch statement in an abstract factory method is appropriate. An abstract factory is one

```
public Student newStudent(int classification)  {
  switch (classification) {
    case UNDERGRAD:
      return new Undergraduate();
    case GRADUATE:
      return new Graduate();
    case DISSERTATOR:
      return new Dissertator();
  }
}
```

Figure 4.7: False Positive for Switch Statement

$badSwitch = \emptyset$
for each node in the AST
node is a switch statement
for each case in switch
    if a new expression is not a child node of statements' case
        $badSwitch = badSwitch \cap node$

Figure 4.8: Algorithm to detect switch statement

of design patterns defined by the Gang of Four [34]. As shown in Figure 4.7, method `newStudent` creates a new instance of Student based on the given classification which is perfectly legitimate.

Our goal here is to reduce the number of false positives. Determining which switch statement is bad is challenging. To avoid mistakenly detecting an abstract factory as bad use of a switch statement, this work ignores those that contain new expressions. An algorithm to detect a switch statement is trivial but is shown in Figure 4.8 for completeness.

4.4.2 Refactorings for Switch Statement

Fowler has defined two composite refactorings called Replace Conditional with Polymorphism and Replace Type Code with Subclasses. Such refactorings are to be used to remove bad switch statements. For Replace Conditional with Polymorphism, a list of statements in each case are extracted to create an overriding method in the corresponding subclass. A switch statement is removed and the original method is made abstract. Nonetheless, it may be necessary to first apply Replace Type Code with Subclasses, if no subclasses are defined. This refactoring creates a subclass for each case in switch statement. Figure 4.9 illustrates the code after applying both refactoring in attempts to remove a switch statement from Figure 4.6.

It is worth noting that the design may not be optimal after the switch statement is removed. More transformations may be necessary depending on the body of each case in the switch statement. For instance, the code in Figure 4.9 can be further improved by adding a new field `fee_per_credit` which leads to duplicates in `getTuition` method. Such methods can be move to class `Student` as discussed in Section 4.1. The final code after the transformations is depicted in Figure 4.10. This research only attempts to remove a switch statement that is used in place of polymorphism. We do not intend to provide the algorithm that will yield an optimal design. Further improvements on the code's structure are left to the developer to carry out himself.

```
abstract class Student {
  static double segregated_fee = 300;
  private int num_credits;

  abstract public static double getTuition();
}

class Undergraduate extends Student {
  ...
  public static double getTuition() {
    return segregated_fee + 300 * num_credits;
  }
}

class Graduate extends Student {
  ...
  public static double getTuition() {
    return segregated_fee + 350 * num_credits;
  }
}

class Dissertator extends Student {
  ...
  public static double getTuition() {
    return segregated_fee + 200 * num_credits;
  }
}
```

Figure 4.9: Remove Switch Statement

4.5 Summary

This chapter has described the code smells studied for this dissertation as well as the sequence of refactorings that can be applied to remove each code smell. We also provide algorithms for detecting each smell. In particular, this work uses an AST-based algorithm to detect duplicated code which cannot detect clones created by statement reordering. Data class and switch statement detections are also performed on the AST. As data classes contains only the fields, accessors and mutators, the detection algorithm is rather straightforward. Detecting wrong use of switch statements, on the other hand, is very complicated and difficult. The approach for feature envy detection

```
abstract class Student {
  static double segregated_fee = 300;
  private int num_credits;
  private double fee_per_credit;

  public static double getTuition() {
    return segregated_fee + fee_per_credit * num_credits;
  }
}

class Undergraduate extends Student {
  public Undergraduate(int c) {
    fee_per_credit = 300;
    num_credits = c;
  }
}

class Graduate extends Student {
  public Graduate(int c) {
    fee_per_credit = 350;
    num_credits = c;
  }
}

class Dissertator extends Student {
  public Dissertator(int c) {
    fee_per_credit = 200;
    num_credits = c;
  }
}
```

Figure 4.10: A More Desirable Result

is discussed in the next chapter.

As we mentioned in the beginning of this chapter that this work focuses on a partial set of code smells. Many researchers work on detection algorithms for other smells. For instance, a number of research works [96, 63] discuss how to identify fragments of code in the long method needed to be extracted. Their approach does not just detect the code smell but also scope down the region of problematic code such which is more useful to the developer. It does not only identify where the problem is but it also provides suggestions on how to fix it.

Chapter 5

Metric for Feature Envy Detection

In object-oriented systems, classes group data and related operations within a specific domain concept, and support object-oriented features such as data abstraction, encapsulation and inheritance. Many researchers have proposed metrics to measure object-oriented software qualities. Such qualities include but are not limited to coupling and cohesion. This work focuses on code smells that indicate poor OO designs with respect to coupling and cohesion therefore only measurements of coupling and cohesion will be discussed here. Du Bois and his colleagues [10] provide discussions on how refactoring can improve coupling and cohesion in software systems.

5.1 Coupling Measures

Low coupling between objects is desirable for modular programming. A measure of coupling is useful in identifying an improper relationship between objects. In order to improve modularity and promote encapsulation, the relationship between classes

should be kept to a minimum. The higher the coupling, the higher the sensitivity to changes in other parts of the system. A small premeditated change in a highly coupled system could progress into a long series of unanticipated changes which makes it more difficult to maintain the system.

Previously, coupling is defined subjectively which make it difficult to use in practice. Chidamber and Kemerer [14] was among the first who defined a metric to measure the coupling between objects. Specifically, their metric are called CBO (coupling between objects). According to their definition, two classes are coupled when methods declared in one class use methods or instance variables defined by the other classes. CBO is well known and is widely used in many software industries.

Myer introduced a much more complicated metric. Coupling is defined in six levels. Such coupling levels are used to measure the interdependence of two modules [68]. Page-Jones extends Myer's work by ordering the coupling levels based on their effects on maintainability, understandability and reusability [75]. If two modules are coupled in more than one way, they are strongly connected and are considered to be coupled at the highest level. In addition to the Myer's six coupling levels, Offutt *et al.* added the zeroth level of coupling for modules that are independent [71]. There are many other approaches to measure coupling not discussed in this work.

Many metrics discussed here can be automated but their computations are done on the source code and require the code to be written beforehand. Some metrics can be computed from the design of the software which allow the software qualities to be measured before starting the implementation [51, 87, 50].

5.2 Cohesion Measures

The concept of cohesion is the practice of keeping things that are related together. A good software design should obey the principle of high cohesion. A highly cohesive module is easy to maintain and reuse. Cohesiveness of methods within a class is desirable because it promotes encapsulation. The measurement of disparateness of methods helps identify design flaws. Lack of cohesion in a class implies that it should probably be split into two or more subclasses.

One of the well-known cohesion metrics is introduced by Chidamber and Kemerer [14]. They proposed a metric called lack of cohesion in methods (LCOM). LCOM evaluates the internal cohesion based on method similarity. The method similarity is measured by considering the number of disjoint sets of instance variables used by methods in a given class (access relationship). Let $\mathcal{M}$ be a set of methods $M_i, i = 1..n$ and I_i be a set of instance variables used by method M_i. If $P = \{(I_i, I_j)|I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j)|I_i \cap I_j \neq \emptyset\}$, then

$$LCOM = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

There are several concerns with LCOM. The most serious problem is LCOM measurement is not very discriminating. If a class has LCOM = 0, it could be interpreted in many ways: 1) it is a highly cohesive class, 2) it is not a very cohesive class, or 3) it is a class with no cohesion.

Many researchers define variants of LCOM to overcome the problems in the Chidamber and Kermerer's LCOM [41]. We only discuss one variant suggested by Henderson-Sellers [41] because this metric will be used in the evaluation of our approach. Henderson-Sellers' definition of LCOM is called LCOM*. Perfect cohesion in LCOM* is when all methods access all attributes. Let $\mathcal{M} = \{M_{i=1}{}^{m}\}$ be a set of methods in a class, $\mathcal{A} = \{A_{j=1}^{a}\}$, be a set of attributes, and $\mu(A_j)$ be the number of methods accessing each attribute A_j.

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu\left(A_j\right)\right) - m}{1 - m}$$

Not only does LCOM* address the interpretation issues in its predecessor *i.e.,* LCOM, but it is also easier to calculate.

Briand's RCI used DD-interactions (flow dependence) and DM-interactions (read dependence) to depict relationships among the class members [13]. Cohesion is then computed by dividing the number of actual DD- and DM-interactions by the number of all possible DD- and DM-interactions of the given class.

Zhou proposed a novel graph representation, class member dependence graph or CDMG, to describe the data and control flow relationships among the members of a class [98]. Unlike other approaches, CMDG describes more types of relationships: read access relationships, write access relationship, call relationship and flow relationship. Zhou proposed the cohesion measure DRC using CMDG which he claims can measure the cohesiveness objectively.

Zhou's recent work [97] emphasizes that special methods such as access methods, delegation methods, constructors and destructors have no influence on the cohesion of a class. Hence, they should be excluded from the abstract of a class. The cohesion measures will be masked if special methods are not excluded from the consideration.

LCOM and their variants do not take into account the effects that the special methods may have on the values of cohesion measures. RCI does exclude some but not all special methods *i.e.,* it excludes only constructors and access methods from the calculations. Only CBMC and DRC considers all types of special methods. While LCOM uses method similarity and RCI uses type and attribute reference, DRC uses dependence relationships. Both LCOM and RCI consider attribute reference and/or method invocation but only DRC considers flow dependences. DRC uses dependence relationships. It considers the direction of dependences between methods and attributes (read/write) as well as potential dependences.

5.3 Feature Envy Metric

As discussed in Section 5.1 and Section 5.2, many software quality metrics have been developed but it is not an easy task to decide which one is suitable and can efficiently detect code smells. Despite a large number of existing software metrics, most of them are designed for software quality measurement and are not appropriate for code smell detection.

At first, we intended to use both cohesion and coupling metrics to detect feature envy. The idea is to first calculate cohesiveness of a class. Then, if the cohesion is low,

Package	Class	DRC Cohesion (1)	LCOM*
default	Driver	0.15625	0.2
	Driver.Controller	0	0
	Driver.AbstractDataAccess	0.04082	0
	Driver.FileAccess	0.08000	0
	Driver.DirectAccess	0	0.5
inventory	AbstractOrderController	0.06250	0
	Catalog	0.03571	0
	Item	0.14063	0.10600
	Order	0.03360	0
	OrderItem	0.13889	0.3000
	Warehouse	0.09375	0
inventory.gui	ItemPanel	0.16327	0.33300
	OrderItemDialog	0.20661	0.50000
	OrderItemPanel	0.19008	0.56200
	OrderView	0.10185	0.80400
	OrderView.OrderListModel	0	0
	OrderView.CellRenderer	0.07813	0.77800

Table 5.1: Cohesion Measurements on Instructor's Code

the coupling metric for that class is computed to determine which class it is coupled with. Based on this idea, we have implemented Zhou's DRC cohesion measure [98] because DRC collects and uses more information from a class. Considering that DRC uses such fine-grained information, we believe it will give more precise measures.

However, our experiments have shown that DRC cohesion measure cannot handle inheritance very well. The cohesiveness of a class is very low if it uses most features from its superclasses. According to Briand's unified framework [12], there are two approaches to compute metrics when inheritance is involved. We can either 1) ignore inheritance by excluding inherited members from the analysis, or 2) include inherited members in the analysis. Both approaches have been implemented and tested.

The main problem is that DRC computes class cohesion based on the number

of members in the class. This overgeneralizes the problem. In a way, DRC is too pessimistic when involving inheritance. It favors classes that use features internally while penalizing classes that use inherited features. This metric indirectly discourages code reuse through inheritance which is one of the key features of object-oriented programming.

Table 5.1 shows that many classes are not cohesive based on DRC measurements *i.e.,* cohesion equals 0. All classes whose DRC cohesion are zero were further investigated and it is found that most classes only use features from their superclasses. For instance, `OrderListModel` is a subclass of `java.swingx.AbstractListModel`. It does not define any new fields and defines 3 methods: two of which are get methods and one method make a call to a method defined in its superclass.

Preliminary results have also shown that it is not sufficient to look solely at the value of a class cohesiveness as it may be misleading. A high value is good but a low value does not necessarily means poor design.

In addition to the cohesion measure, the CBO coupling measure by Chidamber and Kermerer was also implemented. Unfortunately, coupling measures, in general, can give you a rough idea about which classes are tightly coupled but they do not tell which part of the code causes the improper coupling. They do not pinpoint where feature envy occurs.

Since cohesion and coupling metrics failed to locate feature envy, we found it necessary to develop a new metric. One of the reasons they are not suitable for the job could be because coupling and cohesion are computed at class level while feature

envy happens at method level. Even though we did not choose to adopt any discussed metrics in the implementation, many metrics are useful and used as a foundation for the new metric. Ideas we apply to the new metric are:

1. exclude special methods *i.e.,* access methods and constructors. It has been reported by many that special methods would mask the real result [98].

2. exclude inherited members from the analysis. The reason we opt to exclude inherited members is because inherited members are considered parts of the class. Including inherited members will needlessly complicate the analysis.

5.3.1 Internal Process

The granularity of the current cohesion metrics is at the class level while Feature Envy needs a metric whose granularity is at the method level. The new metric is developed based on this nature. Feature envy happens when a method of a class make a lot of method calls to another class, which implies that it is more interested in the other class than the one it belongs to. With such characteristics, we find that in order to locate feature envy:

1. metric must be computed on every method of a class,

2. in each method, information about method calls must be collected

The next step is to locate the source of the problem by finding which outside methods are called and which class they belong to. Calls are then grouped by object in order to determine which objects are called more often than others. Once we

identify the most frequently used object, we know where feature envy occurs. Such information is critical because removing feature envy requires moving the problematic code to its new home.

For all named objects (*i.e.,* fields, parameters and local variables) in each method, the number of method calls on such objects are counted. That number is then plugged into the formula in order to be normalized into a range of [0,1]. Normalization makes it easy to determine the severity of feature envy. We learned from existing metrics that the computed values can be vary and it is difficult to compare metric values if they are unbounded. In this formula, weight is given based on how many calls are made on that particular object.

Let m be the number of calls on *obj* inside the method *mtd*, n be the total number of calls (on any object defined or visible) in the method *mtd*, w be the weight and x be the base where $w, x \in (0, 1)$. The implementation in this work uses $w = 0.7, x = 0.3$. The discussion on the values chosen for w and x is provided in section **??**.

$$\mathsf{FeatureEnvy}(obj, mtd) = w\frac{m}{n} + (1 - w)(1 - x^m)$$

The above formula consists of two parts. The first part of the formula computes the percentage of calls which represents how frequent *obj*'s methods is used comparing to other objects. In other words, the first part calculates the severity of feature envy within the method. The higher the percentage the greater the value of the first term. The second part represents the significance of such an envy relative to those in other

methods. The higher the value of m, the less the value of x^m which makes the second term higher. In other words, the second term is introduced to favor objects that are called on more frequently. For instance, consider $m = n = 2$ and $m = n = 4$. In both cases, 100% of calls in the method are on one receiver. However, the latter case should be given more attention because features of such an object are used more frequently. The weight w in this formula serve as how much magnitude you want to give the internal correlation against external correlation. This work uses $w = 0.5$, which means they are given equal significance. x serves as an exponent that maps the value into a [0,1] range. This work uses $x = 0.5$.

After the values of w and x are chosen, we have to set the threshold or cut-off value for feature envy candidates that would be presented to the developer. The purpose of specifying the cut-off value is to show only feature envy that is considered "serious". Each developer may have different opinions or views on the severity of a feature envy. More importantly, the value of the threshold is subjective. This tool uses 0.5 which means an object that were called on a lot *i.e.,* at least one half of total number of calls outside the class will be reported to the users. Table 5.2 shows representative values for feature envy with respect to the number of method calls made inside a given method.

Consider the code in Figure 5.1 which is a real example of students' and instructor's code from a UWM CS course. Two objects `item` and `_itemPanel` are used in method `updateItemPanel`. After the feature envy metric is computed, we have the following results.

w	x	m	n	Feature Envy
0.5	0.5	0	1	0
		1	1	0.75
0.5	0.5	0	2	0
		1	2	0.5000
		2	2	0.8750
0.5	0.5	0	3	0
		1	3	0.4167
		2	3	0.7083
		3	3	0.9375
0.5	0.5	0	4	0
		1	4	0.3750
		2	4	0.6250
		3	4	0.8125
		4	4	0.9688

Table 5.2: Variables in the Metric

```
private void updateItemPanel() {
  Item item = getItem();
  int q = getQuantity();
  if (item == null) {
    _itemPanel.clear();
  } else {
    _itemPanel.setItem(item);
    int inStock = Warehouse.getInstance().getQuantity(item);
    _itemPanel.setInstock(q <= inStock && 0 < inStock);
  }
}
```

Figure 5.1: Feature Envy Candidate

$$\mathsf{CallSet}(item, updateItemPanel) = \{\}$$

$$\mathsf{CallSet}(\_itemPanel, updateItemPanel) = \{clear, setItem, setInStock\}$$

$$n = 6$$

$$\mathsf{FeatureEnvy}(item, updateItemPanel) = 0$$

$$\mathsf{FeatureEnvy}(\_itemPanel, updateItemPanel) = 0.5(3/6) + 0.5(1 - 0.5^3) = 0.6875$$

5.3.2 Choosing w and x

We perform a number of experiments on the values of w and x. At first, we started with $w = x = 0.5$. One obvious problem we found with this set of values is for the case of $m = n = 1$ which represents delegation. In such a case, the computed metric value is 0.75 which is considerably high. However, delegation is a legit method of sending messages between objects and should not be considered as feature envy. Delegation is one of the main contribution to the number of false positives. Therefore, the values of w and x need to be adjusted. Upon thorough investigations, we found that we gave too much significance on the external correlation.

Each time the values of w and x are adjusted, the number of false positives is recorded. We choose those values by looking at the case when the least number of false positives is produced. After updating the values, the number of false reports is reduced by almost 80%. However, the values that we use in this work may not perform well with other sets of source code. We believe that the results may be better if we apply machine learning techniques in adjusting the values of w and x.

5.3.3 Incorporating an Analysis

Unfortunately, looking at the computed value alone is insufficient. We have to determine if an object in question is written within a block of code. It is necessary to determine if an object is written within a block of code because if it is so, the object could be different at different points in the program which means that it is not really a feature envy. If the object is different than the last occurrence(s), we cannot

```
1:  public void setOrder(Order o) {
2:    if (_order != null) {
3:      _order.deleteObserver(this);
4:    }
5:    _order = o;
6:    _order.addObserver(this);
7:    update(o, o);
8:  }
```

Figure 5.2: Non Feature Envy Instance

move the code because it is semantically wrong to combine activities on two different objects and make them a uniform operation for an arbitrary object. Doing so will cause the program to behave differently. Since the metric gathers information based on names not the pointer (or reference), it could not tell whether the object has been redefined. Further analysis is required to reduce the number of false positives.

Let's look at the code in Figure 5.2. The feature envy metric value of 0.7083 for `_order` which seems like a good candidate for feature envy. However, if we look at the code, we will see that `_order` is written on line 5. `_order` before line 5 and `_order` at and after line 5 are in fact different objects. Since they are different, they are not feature envy.

$$\mathsf{CallSet}(\_order, setOrder) = \{deleteObserver, addObserver\}$$

$$n = 3$$

$$\mathsf{FeatureEnvy}(\_order, setOrder) = 0.7083$$

A less obvious example is previously discussed and shown in Figure 5.1. In this example, the metric gives 0.6875 which is considered significant (above the threshold). However, it cannot be assured if `_itemPanel` remains the same objects throughout

```
class Inventory {
  public ItemPanel _itemPanel;
  private void updateItemPanel() {
    Item item = getItem();
    int q = getQuantity();
    if (item == null) {
      _itemPanel.clear();
    } else {
      _itemPanel.setItem(item);
      int inStock = Warehouse.getInstance().getQuantity(item);
      _itemPanel.setInstock(q <= inStock && 0 < inStock);
    }
  ...
}

class ItemPanel {
  public Inventory i;
  public ItemPanel(Inventory j) {
    i = j;
  }
  public clear() {
    i._itemPanel = new ItemPanel(i);     // _itemPanel is reassigned!!
  }
  ...
}
```

Figure 5.3: Maybe Feature Envy

the entire method by just looking at the code in class `Inventory`. Methods `clear`, `setItem` and `setInstock` could be changing `_itemPanel`. Therefore, it is insufficient to perform just intraprocedural analysis. We need to expand our scope and perform further analysis on methods in questions. For instance, method `ItemPanel.clear()` could be assigning a new object to `_itemPanel` as illustrated in Figure 5.3.

5.3.4 Metric Validation

Many researchers have proposed desirable properties of software metrics. Particularly, for a metric to be useful, it must be valid, reliable, robust and practical. Our metric

will be discussed according to properties summarized by Henderson-Sellers [41].

Validity

Henderson-Sellers defines two types of validity: internal and external. Internal validity addresses how well a measure captures the "meaning" of things that we want to measure. Furthermore, a new measure should correlate with the old one. It is evident that our measure relates to other existing measures. A class with many instances of feature envy implies that it is highly coupled with other classes. External validity relates to generalization issues. In other words, the metric must be generalized beyond the samples that have been measured. Generally, external validity cannot be experimentally determined and it can hardly be achieved. Our metric is originally designed for Java. It is not applicable to other programming language paradigms but it can be adapted for use in other object-oriented programming languages.

Reliability

A metric is reliable if it produces consistent results. Consistency involves stability and equivalence. Stability means it is deterministic, in other words produces the same results given the same input. In our case, provided that the same w and the same x are used on the same input, the metric will produce the same results. Moreover, two feature envy instances have equivalent level of severity, if the computed values are the same.

Robustness

Tsai *et al.* [92] define robustness as the ability to tolerate incomplete information. Furthermore, the robustness is determined based on how well it can handle incorrect input. In this case, the information required for the metric is the source code. The only requirement is that the given source code must not have any compile errors. Our metric does not require the whole program so it is robust to some degree.

Practicality

The metric must be informative. Jones believes a useful metric should be language independent and applicable during the early stages of the development process. However, due to the nature of feature envy which happens after the implementation, it is impossible to come up with a metric that determines feature envy during the design phase. Feature envy can only be revealed after the code has been written. Furthermore, the metric should have the capability of prediction. Our metric also provide the flexibility of adjusting w and x depending on the nature of the applications. It provides a guideline from the semantic viewpoint rather than a mere count of something.

5.4 Summary

A new metric for feature envy is discussed in this chapter. The values of our metric can be uniquely interpreted in terms of the severity of the problem. Since the values

computed by our metric are in the range of $[0, 1]$, it is easy to compare a particular value with other values. We also explained that an analysis can be combined with the metric to improve the accuracy of the results.

It is worthnoting that other research works on feature envy detection [94, 72]. Oliveto *et al.* introduces the concept of method friendships [72] which analyzes both structural and conceptual relationships between methods. Other works, though not related to feature envy detection, also attempt to retrieve semantic information from the source code. Some researchers use information retrieval technique called Latent Semantic Indexing to extract identifiers and comments [60]. Others use natural language processing techniques and introduce the LORM metric [28]. A work by Bavota *et al.* proposes a technique to extract class [7] but we believe that their approach can be modified and applied to detect feature envy, since their approach also includes cohesion metric.

Chapter 6

The Framework

This chapter discusses the overall framework and the architecture of JCodeCanine. Some parts of the framework have already been developed by the Fluid[1]group. We start the chapter by discussing the existing components, the architecture of the system, subcomponents of the system and the details of each module.

6.1 Existing Components

There are two main components in this work: Fluid and Eclipse. The Fluid infrastructure is used mainly for program analysis and code transformations. The Eclipse framework is used as a front-end that interacts with the user.

[1]Fluid is a project in collaboration of Carnegie Mellon University and University of Wisconsin-Milwaukee. It provides a tool to assure that the program follows the design intent.

6.1.1 Fluid

Fluid provides a tool to assure that the program follows the programmer's design intent. The developers can run different analyses on their programs. There are a number of analyses that this work uses which include Effects Analysis and Uniqueness Analysis. The analysis framework is set up in a way that a new analysis can be added easily and without too much hassle.

Program analysis is usually done on an intermediate form that represents the program's structure. The representations that are commonly used are graph and tree. Fluid provides tree-based analyses. The internal representation (IR) which used in Fluid consists of nodes and slots. A node can be used to represent many kinds of objects but this work refers to a node in an abstract syntax tree (AST). A slot can store a value or a reference to a node. A slot can be attached to a node by using an "attribute" or can be collected into a "container". For instance, a method declaration node has an attribute "name" that holds the name of the method and is associated with a container that holds references to other nodes (its *children*) in the AST *e.g.,* a list of parameters, a return type and a method body.

Every analysis in Fluid will be perform on the IR. However, since Fluid IR is an internal representation that the developers do not usually understand, we need a way to obtain the source code back from the IR. This process is called *unparsing*. After the transformations are performed on the IR *i.e.,* refactorings, the unparser is used to obtain the source code which is then displayed to the user.

6.1.2 Eclipse

Our requirement is to develop a tool that works inside an IDE. We choose Eclipse because it is one of the most popular IDEs for Java. It is also easy to extend via a plug-in which eliminates the need to develop the whole user interface from scratch.

6.1.3 Incompatibilities between Fluid and Eclipse

There are a number of incompatibilities between Fluid and Eclipse which make the implementation difficult. One issue involves different representations of the Fluid Abstract Syntax Tree (FAST) and Eclipse Abstract Syntax Tree (EAST). The FAST is more fine-grained than the EAST. On the Fluid's side, the Java source adapter has been implemented to address such a conflict. The Java source adapter, as its name implies, adapts the EAST into the FAST. It basically converts the abstract syntax tree obtained from Eclipse into a different abstract syntax tree with Fluid IR. The adapter allows us to perform different analyses on the Fluid side, since all analyses expect the FAST. The other and more serious issue is concerned with versioning. While Fluid is versioned, Eclipse is not. In other words, Fluid keeps track of changes made in different versions but there is no versioning system in Eclipse. To make them work together we need a mapping mechanism from non-version to version space and vice versa. Eclipse has no knowledge of which system's version (under the Fluid's context) it is working on. Hence, we need a *bridge* to administer the communication between Eclipse and Fluid. The bridge handles everything that involves Fluid versioning system. Its main duty is to keep Fluid and Eclipse synchronized on resource changes.

Fluid	Eclipse
Versioned Fluid AST	Unversioned Eclipse AST

Table 6.1: Fluid and Eclipse Incompatibilities

6.2 Implementation

Section 6.1 describes several attributes of Fluid and Eclipse that pre-exist and serve as foundations of the current work. This section provides description of newly developed components and how they fit in the existing framework.

We developed a tool called JCodeCanine which is an Eclipse plug-in. The main features of JCodeCanine is detecting code smells on Java programs, suggesting different refactorings to the developers and allowing them to apply the suggested refactorings that will remove such smells. Though the implemented tool is for Java programs, the ideas behind this work can be adapted for other object-oriented programming languages.

6.2.1 The Architecture

JCodeCanine's components can be divided into three groups depending on where the activities are taken place. The Eclipse group is the front-end which provides the user interface, interacts with the users and handles all user actions. The Fluid group contains back-end components that involve with IR nodes, versioning and analysis. The last group, the *in-betweener*, consists of those that provide the interconnections between Eclipse and Fluid.

- **Eclipse's side**: Editor, User Action Handler
- **Fluid's side**: Code Smells Detector, Refactoring Manager, Annotation Suggester
- **In-betweener**: Java Source Adapter, Bridge, Promise Parser, Unparser

Figure 6.1 shows the architecture of JCodeCanine. The process starts when the developer invokes code smells detection through Eclipse. Eclipse parses the Java source file into the EAST. The EAST then gets adapted into the FAST by the Java source adapter. After the FAST is obtained, a new version is created. The code smell detector performs analysis on the FAST, marks the region of problematic code and returns the information in the form of Eclipse's warning markers. Each marker is linked with the resolution for the problem *i.e.,* refactorings. If the developer chooses to fix the code smell, the responsibility is shifted to the refactoring manager which performs the semantic check and takes care of the code transformations. Since the analyses and the transformations are done on the Fluid IR, we need the *unparser* to translate the IR nodes back to Java source code. Currently, the code smells detector is executed whenever a resource is changed which could be a double-edged sword. On one hand, it is automatic and convenient since it shows immediate results to the programmers. On the other hand, this approach may not be applicable for large software systems considering our resource consuming implementation.

If the developer doesn't change anything since the last time the detector is run, there is no need to re-compute analysis information.

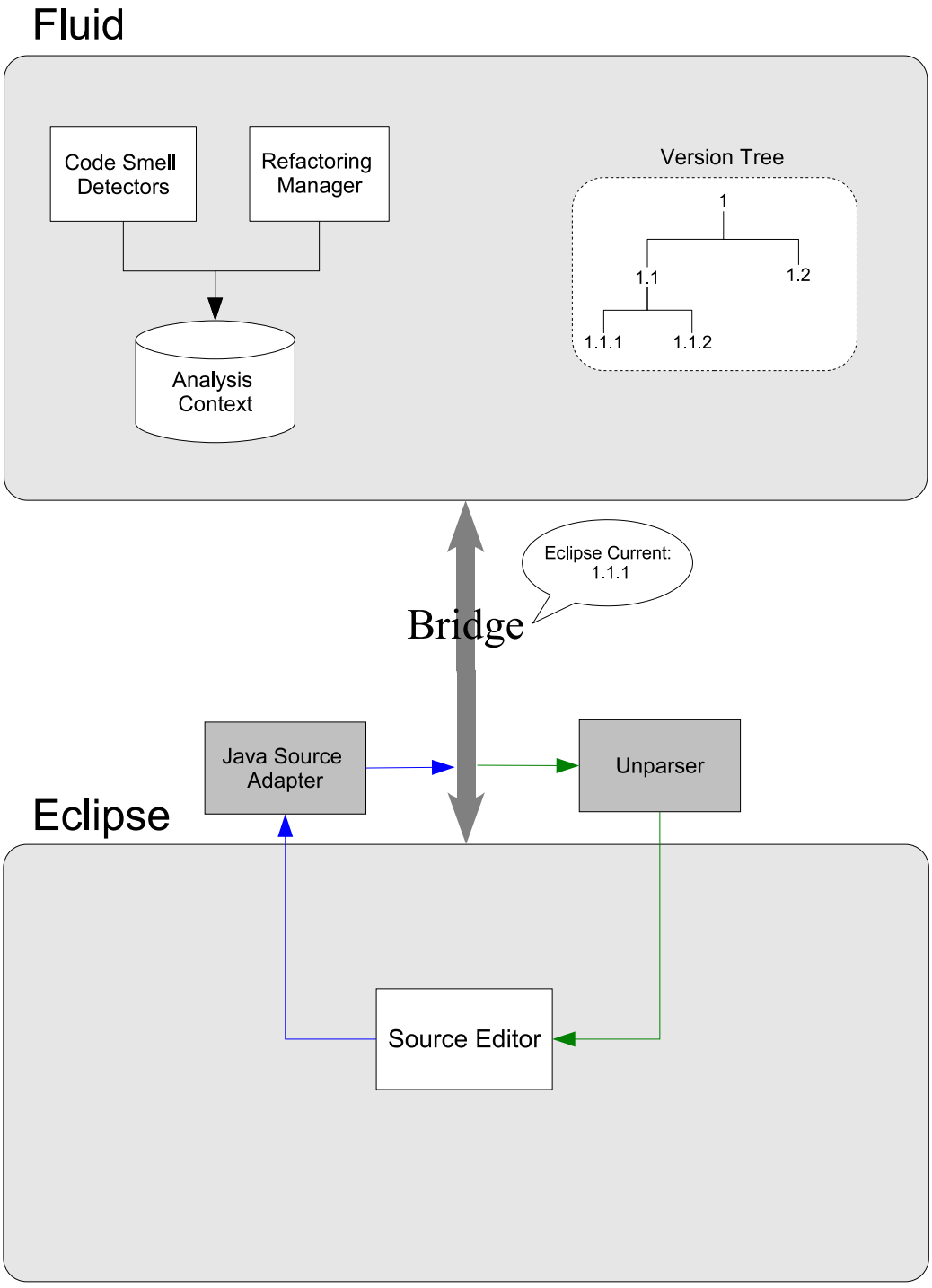


Figure 6.1: Architecture of JCodeCanine

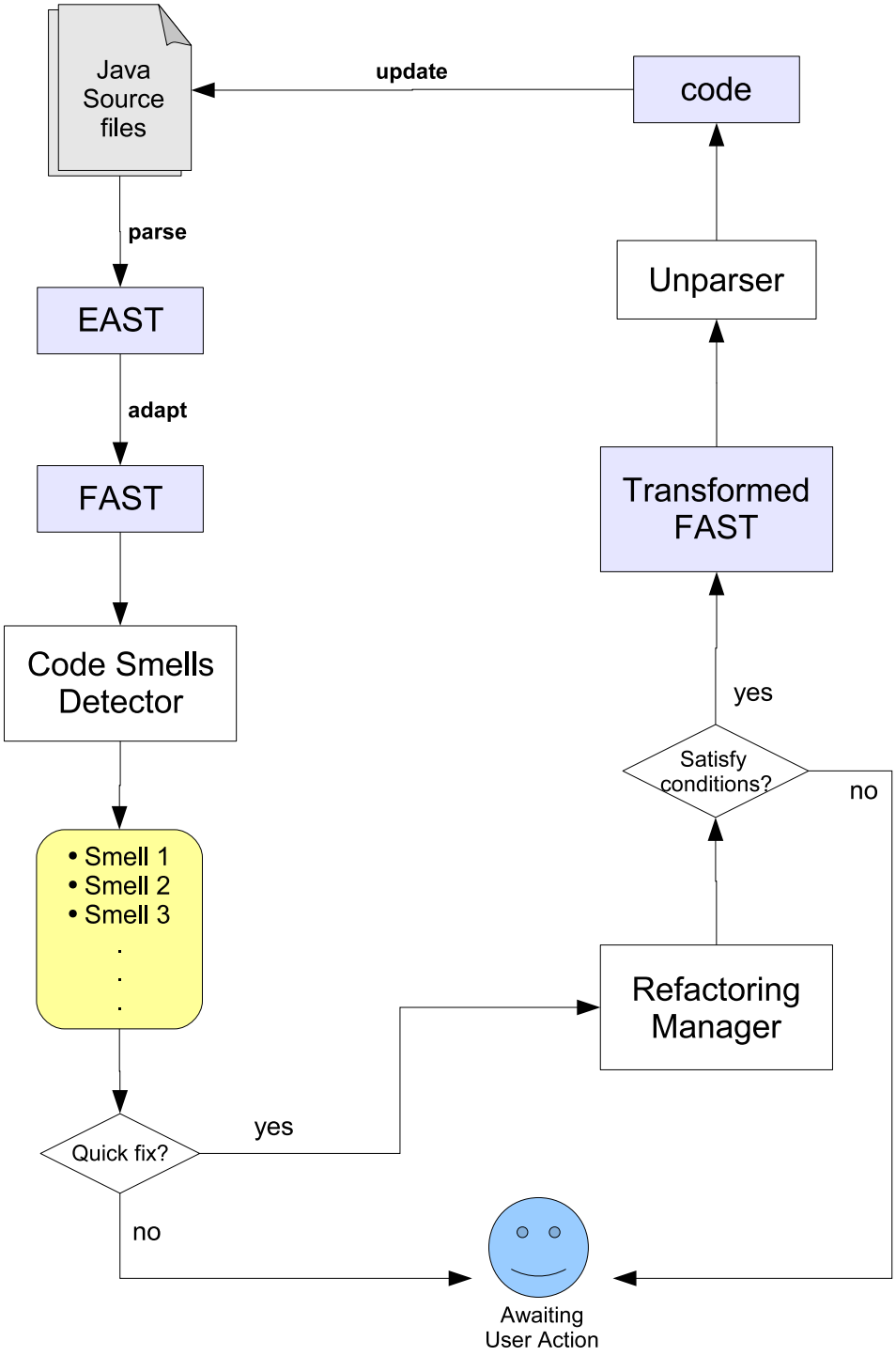


Figure 6.2: Process Cycle in JCodeCanine

6.2.2 Refactoring Manager

In JCodeCanine, the programmer can refactor his code either through a quick fix after code smells are detected, or through the menu directly without checking for code smells. After a refactoring request is invoked, the source code is adapted into Fluid IR by the Java source adapter. The semantic preconditions for that particular refactoring are checked. If one of such conditions is not satisfied, the code will not be refactored. One distinction between this work and Eclipse's refactoring is that semantics check are taken into consideration before attempting any refactoring. Using refactoring support in our tool will not introduce compile errors. Hence, we consider our approach to be true behavior preservation. Eclipse refactors the code first and lets the compiler catch any errors that may occur. In a way, Eclipse's refactoring support is not behavior preserving.

Each refactoring has an inverse operation *e.g.,* Extract Method and Inline Method. In general, refactoring can be undone by applying its inverse refactoring. In contrast to other approaches, our refactoring tool utilizes versioning in Fluid for undoing/redoing refactoring. When a refactoring is applied, the program snapshot is captured as a version in Fluid. Hence, undoing/redoing is as easy as switching back and forth between versions or traversing up and down the version tree. The bridge then informs Eclipse of the current version that it is working on. One benefit gained from using a versioning model for undo/redo is that the semantic condition check that needs to be performed before any refactoring can be skipped. Unfortunately, the Fluid framework is not small. Capturing a version whenever the code is refactored

can be space intensive. The trade-offs between time and space will have to be weighed out by the developer.

6.2.3 Code Smells Detector

Code smells detector acts as a sniffing dog. It checks and determines which part of the code is stinky using analyses discussed in Chapter 4. In this implementation, one detector is developed for each code smell. Basically, the detectors are invoked in a sequence. Each detector analyzes the IR nodes and when it detects a code smell, the problematic nodes are marked. The detector keeps track of those marked nodes. When all code smells are detected, the IR is unparsed into Java source code. The region of source code that is from the marked IR nodes is then highlighted with a "marker".

6.2.4 Code Smells Resolution

As discussed in Chapter 4, each code smell is resolvable using one refactoring or a series of refactorings. The implementation follows that same idea *i.e.,* each code smell is linked with a resolution. Basically, the code smells resolution is the process that attempts to remove the detected code smells by means of refactorings. The refactoring process is then taken care of by the refactoring manager as usual.

6.2.5 Annotation Suggester

The Fluid analysis framework requires the program to be annotated and such a process can be overwhelming to any developer. The annotation suggester relieves some burden off the programmer by giving an advice of what kind of annotations is required. Our annotation suggester is preliminary. It only considers a portion of code in question and suggests annotations that allow a specific analysis to be executed. For instance, if uniqueness analysis is being executed, it will only suggest those that are related to uniqueness analysis. If effects analysis is later executed on the same method, the programmer is required to add effect annotations. This on-demand approach could be unpleasant since the programmer may spend most of their time annotating the code or worse, they could end up annotating the whole program. Specifically, if a message chain exists, it is necessary that all methods in the chain are checked which means that they must be annotated. To avoid this nuisance, we only check the immediate message and assume that other methods in the chain do not violate the conditions. We are aware that it would be beneficial to adopt "annotation inference" where annotations are automatically derived from the code. However, annotation inference is beyond the scope of this research.

6.3 JCodeCanine's Key Features

This section demonstrates various scenarios when using JCodeCanine. First, we show various code smells detected by JCodeCanine and how to automatically remove code

smells with suggested refactorings. Then, we exhibit the situation when refactoring is invoked by the developer.

6.3.1 Code Smells Detection and Resolution

Code smell detectors are executed when a resource changes. Each code smell detected is shown as a warning message. The tool provides the following information for each smell:

- type of code smell,
- file name, and
- location of code smell in the source (line number).

Figure 6.3 demonstrates how the tool displays the detected code smells to the user. Each marker links to the source location. The problematic code is highlighted with a yellow line which acts as a warning sign to the developer.

In addition to the general information about the code smell, each smell marker is linked to an automatic resolution. The user has an option to remove the detected code smell through a *quick fix* (see Figure 6.5). By invoking a quick fix, the suggested refactorings are automatically applied to the code.

6.3.2 Stand Alone Refactoring

Apart from an automated code smells resolution, developers also have an option to initiate and perform refactoring through the menu.

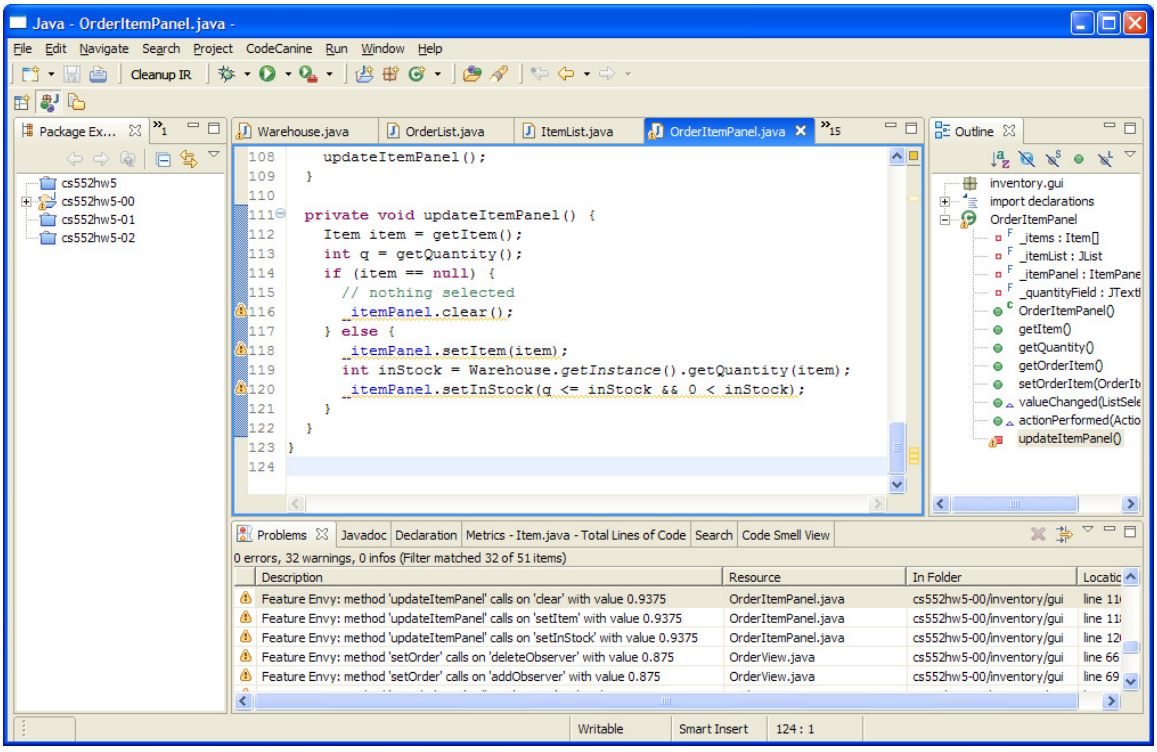


Figure 6.3: A Snapshot of Feature Envy Detected

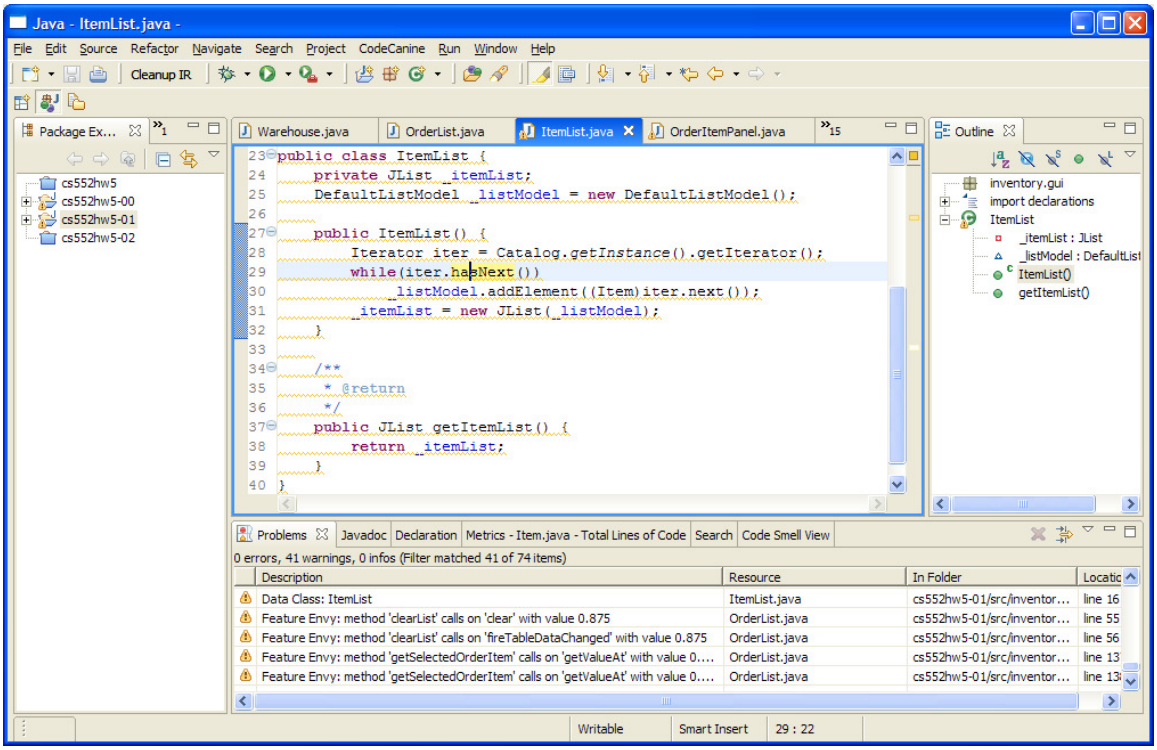


Figure 6.4: A Snapshot of Data Class Detected

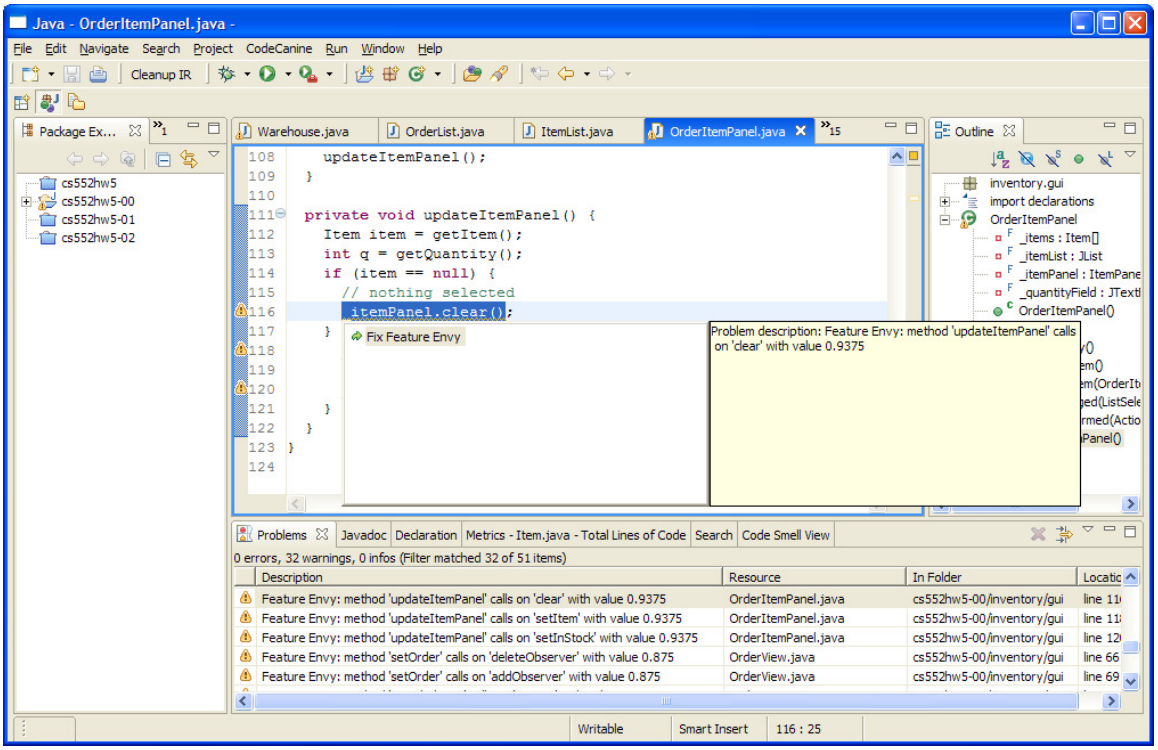


Figure 6.5: Apply Refactoring through Quick Fix

After a refactoring request is initiated, the tool checks the semantic conditions behind the scene. Figure 6.6 shows the situation when the chosen refactoring cannot be applied because it violates our behavior preserving conditions. Particularly, the programmer wants to apply rename the `bought()` method in the `IPhone` class to `updateStock()`. However, apply such a refactoring will create an overloading method and change the behavior of the `updateStock()` method from increasing the quantity by one to decreasing it by one. JCodeCanine catches that this serious side effect and report the issue to the programmer.

```
public class Product {
  void updateStock() {
    ++quantity;
  }

  private int quantity;
}

public class IPhone extends Product {
  void bought() {
    --quantity;
  }
}
```

Figure 6.6: Semantic Conditions Violated if `bought()` is Renamed to `updateStock()`

6.3.3 Annotation Suggestions

While the Fluid analysis framework requires programs to be annotated, JCodeCanine will try its best in performing the analysis with available information. If it does not have enough information, it will ask the programmer to annotate their code. JCodeCanine does have the ability to suggest annotations but it may not comply with the design intent. The developer is responsible for providing correct annotations in order to obtain the intended results.

Figure 6.7 shows the portion of code that annotation suggester will be activated when the developer wants to move a field `warrantyExpirationLeftFrontTire` to a component class `Tire`. Since this particular refactoring requires uniqueness analysis and no annotation has been declared, JCodeCanine advices the developer to add an annotation `@unshared` to the field which indicates that such a field is unique and cannot be shared among objects. After the expected annotation is added, the semantic condition check is resumed by performing a uniqueness analysis. If it passes

```
public class Automobile {
  Engine autoEngine;
  int numOfPassenger;
  Tire leftFrontTire;
  Tire rightFrontTire;
  Tire leftRearTire;
  Tire rightRearTire;
  WarrantyInfo warrantyExpirationLeftFrontTire;
  WarrantyInfo warrantyExpirationRightFrontTire;
  WarrantyInfo warrantyExpirationLeftRearTire;
  WarrantyInfo warrantyExpirationRightRearTire;
  ...
}

public class Tire {
  ...
}
```

Figure 6.7: Scenario where Annotation Suggester is Invoked

the uniqueness analysis *i.e.,* the field `warrantyExpirationLeftFrontTire` is unique, then the field will be moved to the destination class which is the *Tire* class.

6.4 Summary

The main framework of JCodeCanine is discussed in this chapter. JCodeCanine is implemented based on existing components which are Fluid and Eclipse. The Fluid's infrastructure is used as the main component for back-end process and Eclipse is used mainly for the user interface. Though Fluid and Eclipse are incompatible in various aspects, the in-betweeners like the bridge and Java source adapter provide the harness between the front and the back end. The key modules *i.e.,* Code Smells Detector, Refactoring Manager and related analysis are developed on the Fluid's side to show that our work is sound. The nature of our Fluid analysis framework is that it requires

the developers to annotate their design intents. JCodeCanine does have the ability to suggest annotations but it currently is considered experimental. It is possible to write a more sophisticated annotation suggester or to implement a mechanism for annotation inference which could reduce the burden on the developers but such a system is beyond the scope of this work.

Chapter 7

Empirical Results

Murphy-Hill and Black present seven habits that an effective smell detector should possess [67]. Such seven habits include:

- *Availability*: A smell detector should not require much effort from the programmer. Smell information should be made available to the programmer as soon as possible.

- *Unobtrusiveness*: A smell detection tool should perform its job without interfering the programmer while he is coding.

- *Context-Sensitivity*: A smell detector should only point out smells that are related to the current programming context.

- *Scalability*: A tool should not overwhlem the programmer with smell information if a significant number of smells is identified.

- *Relationality*: A smell detector should show relationships between code fragments that contribute to code smells.

- *Expressiveness*: A smell detection tool should provide explanation on why the smell exists.

Since our approach is to develop and integrate code smell detection tool with Eclipse IDE, JCodeCanine is context-sensitive, relational, and expressive. The smells found by JCodeCanine are reported with details in the problem view. Each message in the problem view is also linked to the code where the programmer can double click and see the source of the problem in the editor. All code fragments that cause code smells are also underlined in the editor view.

According to Murphy-Hill and Black's definitions, JCodeCanine is not available and obtrusive because the analysis will be performed only when the programmer chooses to start the smell detector. When the smell detector is running, the programmer is not able to do any coding. Code editing is blocked during this process. In our opinion, JCodeCanine is not so scalable because it shows all smells that it detects. We believe that allowing the programmer to apply filter to the smells list will help on the scalability issue. However, this issue is yet to be explored.

After we have developed code smells detection with refactoring support tool for Java program, we evaluate our approach by determining the soundness of the code smells detection. We also look at the number of suggested refactorings and determine if they are sound both syntactically and semantically.

We perform tests on various sizes of code. Our test cases include:

- UWM CS552 homeworks written by students and the instructor which include homework 4, 5, 6 and 7,
- `java.util` package (JDK 1.4.2),
- `java.lang` package, and
- `fluid.util` package.

The size of each CS552 homework is about 500 to 3K lines of code. Each homework instance is tested separately. However, we aggregate the results by homework for preciseness. The `java.util` package consists of 123 classes and 13K lines of code. The `java.lang` package consists of 100 classes with 10K lines of code. The `fluid.util` package contains 106 classes with 50K lines of code.

7.1 Code Smells

In order to determine how well the code smells detection component performs, we look at the number of code smells detected. The number of false positives is also recorded to further reflect the accuracy of our approach. False positives, in this context, are the legitimate code that are detected as code smells. The comparison of these two numbers give us an idea of how accurate our algorithms are. As shown in Table 7.1 that the average accuracy of detection all four code smells is at 54%.

The results from Table 7.1 show that our duplicated code detection cannot find any code clones. We have investigated on why its performance is rather poor and found

Code Smell	Detected	False Positives	Accuracy (%)
Duplicated Code	13	4	69%
Feature Envy	24	2	92%
Data Class	5	0	100%
Switch Statement	37	30	19%

Table 7.1: Numbers of Code Smells Detected, False Positives and Accuracy

out that the homework code instances do not contain any duplicated code. Regarding `java.lang` and `java.util` packages, we cannot go through the code thoroughly due to their sizes. However, based on our rough observation, both java packages are quite well written and do not contain duplication structurally. We also performed tests on our duplicated code detection by creating new packages with known duplicates and our detector can correctly identify those clones. The limitation of our detection algorithm is similar to any AST-based clone detections in such a way that it cannot detect clones which are from statement reordering.

Regarding false positives for feature envy, data class, and switch statements, the tool reports 46% overall false positives: 8% false positives for feature envy, no false positives for data class and 81% false positives for switch statements.

We further determine the reason why the tool returns such high percentages of false positives. This process has to be done manually. After looking at each and every instance, we found that:

- Our work can successfully detect feature envy.
- Falsely detected data class can be categorized into: 1) a subclass of Exception and Error classes. 2) a real data class by the programmer's intents e.g.

`java.util.CurrencyData`.

- Poor use of switch statements is very hard to detect in general. We need to find a new heuristic algorithm that can correctly determine the poor use of switch statements without introducing a lot of false positives.

The results, though not as satisfying as expected, are very informative. Currently, the algorithms for data class and switch statements are syntax-based. We speculate that program's semantic analysis may be able to reduce the number of false positives. However, such an idea has yet to be investigated in the future.

Another aspect for evaluating the accuracy is testing for false negatives. False negatives in this context are real code smells that are left undetected by the system. In order to test for false negatives, test programs with known code smells are created. Then, we run our detectors on those test programs. The detectors performed really well as it can locate all instances of code smells. No false negatives are found.

7.2 Refactorings

In this section, we are particularly interested in seeing the quality of refactorings suggested by our system. Here, the number of suggested refactorings with respect to syntactic correctness and semantic preservation is measured. By looking at these numbers, we are able to determine whether behavior preservation can be realistically achieved in the automated refactoring tool and what the difficulties are. The three categories of refactorings we have measured are listed below.

Project	Type 1	Type 2	Type 3
hw4	4	4	2
hw5	5	3	2
hw6	9	9	6
hw7	13	11	9
`JCodeCanine`	15	15	10
`fluid-eclipse`	20	16	9
`fluid`	58	51	34

Table 7.2: Number of Suggested Refactorings

1. total number of refactorings suggested by our code smells detector (Type 1)

2. refactorings from 1) that do not break compilation (Type 2)

3. refactorings from 2) that are semantics-preserving. (Type 3)

Type 2 refactorings are syntactically sound. They are refactorings that will not cause compile errors in general. However, behavior preservation is not guaranteed if applying type 2 refactorings as is. On the other hand, type 3 refactorings are both syntactically and semantically sound. This type of refactoring are safe to be applied and will neither break the code nor change the program's behavior.

Table 7.2 shows the breakdown number of each type of refactorings that are suggested by JCodeCanine. Out of total refactorings suggested by JCodeCanine, 92% are safe syntactically and 28% are safe syntactically and semantically. Ideally, we would want the number of type 1, type 2 and type 3 refactorings suggested by our system to be equal. However, such figures are reasonable since not all code smell detection algorithms include semantic analysis. After examining those suggested refactorings, we also notice that one of the obstacles is the existing structure of the program especially

variable and method naming. The program may need to be restructured or refactored first in order for the suggested refactoring to be applied correctly. Furthermore, a number of suggested refactorings that are unsound or unsafe are attributed by the existence of false positives in code smells detection.

7.3 Code Qualities

In this section, we analyzes the performance of our tool by comparing the quality of the code before and after running our tools on each test package. In order to make sure that the program semantics are well-preserved, we choose to apply only type 3 refactorings (syntactically and semantically sound refactorings as discussed in Section 7.2.

The metrics used to measure code qualities in this research are mostly from Chidamber and Kemerer's work [14] as they are pioneers in software quality metrics. They have proposed a set of static metrics that are designed to evaluate the quality of an object-oriented design. Their metrics are widely known and used in software development process. This work uses some of their metrics for measurements which are:

1. Weighted Method per Class (WMC)
2. Depth of Inheritance Tree (DIT)
3. Number of Children (NOC)
4. Afferent Coupling (Ca)

5. Efferent Coupling (Ce)

6. Lack of Cohesion between Methods (LCOM*)

7. McCabe's Cyclomatic Complexity (CCN)

A short description of each metric is provided in Appendix A.

Table 7.3 shows the objectives for each software metric according to Rosenberg [77]. For some metrics, the lower number the better; however, some metrics are considered trade-offs between readability and complexity. We measured the software quality before and after applying suggested refactorings using Chidamber and Kermerer object-oriented metrics and the comparisons are shown in Table 7.4, Table 7.5 and Table 7.6. Values presented in these tables are the average. The metric calculations are from several opensource Eclipse plug-ins *i.e.,* `metrics.sourceforge.net` [83], Google's CodePro Analytix [36] and Analyst4j [16] which provide an extensive numbers of software quality metrics. Note that this work only considers and analyzes metrics that are related to the code smells discussed in Chapter 4; hence, we do not discuss all metrics here. We also summarize the impacts of refactorings on the value of each metric where + means positive impact, - means negative impact and = means no improvement after refactorings have been applied in Table 7.7. Our results show that, in most cases, applying refactorings make positive impacts with respect to object-oriented metrics. Refactorings that make negative impacts include Extract Method as it increases the number of methods in a class; therefore, WMC is increased after refactoring. Other metrics that receive negative impacts include DIT

Category	Metric	Granularity	Objective
Complexity	WMC	Class	Low
Size	DIT	Class	Trade-Off
Size	NOC	Class	Trade-Off
Coupling	Ca	Class	Low
Coupling	Ce	Class	Low
Cohesion	LCOM*	Class	Low
Complexity	CCN	Method	Low

Table 7.3: Objectives for Different Metrics

Project	LOC	WMC		Ca		Ce	
		Before	After	Before	After	Before	After
hw4	558	14.500	14.500	1.333	1.000	0.500	1.000
hw5	1,167	12.941	12.801	2.000	1.667	1.667	1.333
hw6	1,870	9.914	9.080	4.800	4.600	4.400	4.200
hw7	2,632	10.022	9.984	5.333	5.000	4.000	3.750
`JCodeCanine`	9,729	10.830	10.992	8.000	7.677	6.500	6.000
`fluid-eclipse`	17,431	37.763	37.940	8.615	8.512	5.077	4.922
`fluid`	223,041	18.578	17.659	122.012	118.236	28.549	25.489

Table 7.4: Comparison of Software Metric Measurements: Low (1)

and NOC. As we have discussed earlier that the values of these two metrics are trade-offs between readability and complexity. The higher values do not necessarily mean poor quality. It depends on what the developers actually focus on.

7.4 Discussion

Our results concur with Du Bois *et al.* [10] that code smells with respect to improve coupling and cohesion are sparse and difficult to find. In addition, we find that most code smells are difficult to find in general. The fact that our metric can detect a

Project	LOC	LCOM*		CCN	
		Before	After	Before	After
hw4	558	0.214	0.210	2.000	2.000
hw5	1,167	0.271	0.271	1.583	1.583
hw6	1,870	0.204	0.204	1.684	1.684
hw7	2,632	0.224	0.224	1.733	1.667
`JCodeCanine`	9,729	0.161	0.153	1.880	1.760
`fluid-eclipse`	17,431	0.215	0.207	2.721	2.679
`fluid`	223,041	0.115	0.108	1.834	1.710

Table 7.5: Comparison of Software Metric Measurements: Low (2)

Project	LOC	DIT		NOC	
		Before	After	Before	After
hw4	558	3.250	3.250	0	0
hw5	1,167	2.941	2.941	0.118	0.118
hw6	1,870	3.029	3.029	0.371	0.371
hw7	2,632	3.022	3.022	0.5	0.5
`JCodeCanine`	9,729	2.415	2.502	0.279	0.289
`fluid-eclipse`	17,431	1.753	1.753	0.280	0.280
`fluid`	223,041	3.904	3.972	0.780	0.792

Table 7.6: Comparison of Software Metric Measurements: Trade-off

Project	WMC	DIT	NOC	Ca	Ce	LCOM*	CCN
hw4	=	=	=	+24.981%	+50.000%	+1.869%	=
hw5	+1.082%	=	=	+16.650%	+20.036%	=	=
hw6	+8.412%	=	=	+4.167%	+4.545%	=	=
hw7	+0.379%	=	=	+6.244%	+6.250%	=	+3.808%
`JCodeCanine`	-1.496%	-3.602%	-3.584%	+4.038%	+7.692%	+4.969%	+6.383%
`fluid-eclipse`	-0.469%	=	=	+1.196%	+3.053%	+3.721%	+1.544%
`fluid`	+4.947%	-1.742%	-1.538%	+3.095%	+10.718%	+6.087%	+6.761%

Table 7.7: Impacts on Software Quality (+ Positive, - Negative, = No impact)

number of real feature envy instances is very satisfactory.

While measurements are done in different aspects *i.e.,* size, complexity, coupling and cohesion, we focus on values of coupling and cohesion measures as they are the key of object-oriented programming. Results show that coupling measurements decrease while cohesion measurements increase after applying suggested refactorings. Even though the improvements are insignificant, it proves that our approach works and is not trivial. Furthermore, our work improves the design to some extents. Such a claim is made based on tests and an in-depth investigation on our home-brewed application. The reason that we take this approach is because reasoning a program is difficult and time consuming. With an in-house project, the code's intent is clear to us; hence, it is easier to analyze the results.

Nonetheless, we found that, in some cases, the approximation for the analysis does not perform well as our analysis tends to be restricted and too conservative. Any changes that could change or modify the behavior of the program will be disregarded. Further study needs to be done in order to provide more flexibility while preserving the program's behavior which is the main objective of this work. Balancing between these two extremes is a challenge.

The following subsections discuss how each code smell affects the software metrics according to our findings.

7.4.1 Duplicated Code

Intuitively, removing code clones will reduce the number of lines of source code. As discussed in Chapter 4 that in object-oriented programming, clones at different locations must be handled differently. Clones in the same class hierarchy can appear in the same method or in sibling classes. Removing such clones will reduce LOC. However, if the clones are found in two or more unrelated classes where an abstract superclass is needed, LOC and NOC measurements will be increased. Removing duplicated code affects the size-related measures. At this point, it is still obscure whether duplicated code removal has effects on coupling and cohesion measures.

7.4.2 Feature Envy

It is known that feature envy is not desirable in object-oriented programs. Feature envy removal is an attempt to reduce bad coupling. Theoretically, the Ca and Ce metrics should be lower once feature envy instances are removed as they measure the coupling. Our findings are indifferent as the measurements from both metrics are decreased. Furthermore, we also found that remove a case of Feature Envy indirectly increases the class cohesion since the value of LCOM* is lower.

7.4.3 Data Class

Since the number of data classes detected by our tool is not significant, we ran tests on an ongoing software project with 57K lines of code and 375 classes. Our tool found a number of data classes in this project. The result is then shared with the software

developers. They confirmed that some classes were incomplete and some classes were no longer necessary. After applying refactorings on unwanted data classes, the number of classes (NOC) decreased and in the case when the data class inherits from other class, the depth of inheritance tree (DIT) was also reduced. We also noticed that there was a slight improvement in terms of coupling and cohesion when removing a data class.

7.4.4 Switch Statement

Removing switch statement does not have much improvements on the static software quality metrics. On the contrary, the quality appears to be worse when using existing measurements. Since a wrong use of switch statement is usually resolved by creating an abstract superclass, removing a switch statement this way increases the number of classes (NOC) as well as the depth of inheritance tree (DIT). There is no static metric that provides measurements of polymorphism, since polymorphism by its nature can only be observed and measured at run-time. Approaches to dynamically measure polymorphism include 1) calculating the polymorphic behavior index which analyzes internal and external reuse [15], or 2) calculating the number of potentially polymorphic instructions, the number of receiver types (receiver polymorphism) and the number of different target methods (target polymorphism) [23].

7.5 Summary

The evaluation is carried out with respect to the detection accuracy, behavior preservation and the code qualities. In regards to the accuracy of code smell detection algorithms, our system reports a number of false positives. We believe that it is because our analyses are somewhat conservative. However, since this work concerns about behavior preservation, a warning message does not hurt the code because the developer will have a chance to inspect the detected code smells.

To further evaluate the accuracy of the system, programs with known instances of code smells are also tested and there are no false negatives *i.e.,* all known code smells are detected by JCodeCanine. Our results show how semantic preservation should be taken into consideration whenever changes are applied to the existing software system. Though the change does not introduce any compile errors, the developers cannot take any changes for granted. Unsafe refactoring is harmful because tracing for semantic errors or behavioral changes is very difficult. Furthermore, the results also show that our system helps improve internal code qualities to some extent. It could detect code smells and suggest proper refactorings. Even though the improvements are not significant, they are promising. We also found that some metrics are too vague and could not provide clear quality measure for object-oriented programs. In fact, applying some refactorings could make the quality worse according to some software metrics.

Chapter 8

Conclusion

Previous work has introduced many approaches to find refactorings [49, 91, 19]. Some work mentioned the thoughts of combining code smells detection with refactoring tool but none has actually implemented it.

Many researchers rely on the metrics alone to detect code smells or find refactoring opportunities. Our work has shown that an analysis can be used in addition to metrics which results in a more precise result. Not only does this dissertation introduce a metric for feature envy, it also proposes a novel approach by demonstrating how an analysis can be integrated into a metric which allows us to obtain a measurement from the semantic viewpoint.

This work proposes and develops a framework that combines the processes of identifying and applying refactorings. Such a framework allows information from analysis to be reused and avoid recomputing any information we already know. Therefore, it helps improve the efficiency of the overall system.

This work shows that integrating the two processes give us more satisfiable results. It forces us to look at the problem in a big picture. Program analysis plays a crucial part in this work. Even though JCanine only supports Java code, the idea of integrating refactorings with code smells detection can be applied to other object-oriented programming languages.

Last but not least, this research has contributed:

- analyses required to check semantic preconditions for the chosen refactorings.
- analyses required to detect each code smell discussed in this thesis.
- relationships between code smells and refactorings (in terms of analysis used)

Our current implementation can be improved in many ways:

- *Persistence*: This work does not use persistence. However, Fluid has a mechanism where versioned information can be stored and re-loaded. Regenerating versioning information is space intensive. The system would be more efficient with persistence.
- *Incremental Binding*: With incremental binding, the binding information does not need to be recomputed and can be derived between versions.
- *Incremental IR Updates*: Daniel Graves [38] implemented an incremental updater which allows us to reuse some existing nodes. Unlike the traditional method where the whole FAST is regenerated everytime the source is adapted into IR nodes, an existing FAST is incrementally updated from an EAST. New

IR nodes are generated only when necessary. When comparing general updater's and incremental updater's memory usage, differences of 72% - 78% are observed. Incremental updater is significantly more efficient than general updater.

- *Unparser*: The unparser does not pretty print the code. It could be improved so that the user's code formatting is preserved. Preserving the original code formatting after unparsing the internal representation is quite difficult in general.

Other issues that are still open for future research include:

1. *Adaptive Thresholds*: Since different types of applications may require different thresholds for the metric, it is plausible to adapt machine learning techniques that could assist us in choosing a more suitable threshold values [53, 57].

2. *Dynamic Measures for Semantic Information*: Since our main concern is to obtain quick analysis on the source code, analyses in this work are structural-based. Using only structural analysis is found to be insufficient if accuracy is the goal. We believe that incorporating our approach with dynamic semantic metrics will yield better results. However, the main challenge is how to balance the accuracy with the overhead introduced by dynamic metrics or measurements.

3. *Annotation Inference*: Currently, the programmers are required to annotate their programs which can be burdensome. It would be more user-friendly, if annotations are automatically inferred from the source code.

4. *Performance Evaluation for Refactored Programs*: It would be interesting to see how refactorings affect the program performance and whether better designs comes with the sake of the performance.

Appendix A

Software Metrics

Depth of Inheritance Tree (DIT) is the maximum length from the node to the root of the tree. Deeper trees constitute greater design complexity, since more methods and classes are involved.

Number of Children (NOC) is number of immediate subclasses subordinated to a class in the class hierarchy. The greater the number of children, the greater the reuse. However, if a class has a large number of children, it may be a case of misuse of subclassing.

Weighted Methods per Class (WMC) calculates the sum of cyclomatic complexity of methods for a class. A low WMC indicates high polymorphism while a high WMC signifies a complex class.

Afferent Coupling (Ca) computes the number of classes from other packages that depend on classes in the analyzed package (incoming dependencies).

Efferent Coupling (Ce) counts the number of types of the analyzed package depending types from other packages (outgoing dependencies).

Lack of Cohesion in Methods (LCOM*) measures the relative disparateness of methods in the class. Cohesiveness of methods within a class is desirable because it promotes encapsulation. The measurement of disparateness of methods helps identify design flaws. Furthermore, lack of cohesion implies classes should probably be split into two or more subclasses. Many researchers define variants of the original LCOM to overcome the problem with the original metric from Chidamber and Kermerer. The measurements in this section is one variant of LCOM by Henderson-sellers [41], also known as LCOM*.

McCabe's Cyclomatic Complexity measures a number of linearly-independent paths through a program module [62]. McCabe suggested that the cyclomatic complexity should not be greater than 10.

Appendix B

Case Studies

The code used in this section is written by students and instructor in CS 552 class for their homeworks. The tool is run on students' code and following is our findings in the early stage of testing.

B.1 Definitely a Feature Envy

This is the case when we can determine that the suspicious code is indeed feature envy without looking at other parts of the code. Figure B.1 illustrates an instance of feature envy where the programmer should consider moving `_index` to the class that the *put* method is defined in.

```
public Object put(Object key, Object value) {
  if (!value instanceof Item) {
    throw new IllegalArgumentException(``...'');
  }
  Object old super.put(key, value);
  if (old != null)
    _index.remove(old);
  _index.add(value);
  return old;
}
```

Figure B.1: Feature Envy Instance

B.2 Calls on Objects of the Interface Type

Calls on object of the interface type are complicated and difficult to handle. Though an interface is a type just like a class is, they are different. Unlike a class, an interface only defines and never implements methods. Classes that implement the interface are responsible for implementing the methods defined by the interface. With this nature of the interface, we cannot move the implementation into the interface. The approach used in the previous section cannot be used here.

One way to handle this case is to convert the interface to an abstract superclass. Since it has become a class, the implementation can be moved into it. However, these steps cannot always be done because there could be more than one classes that implement this interface. If every class that implements the interface is on the same hierarchy, we are home free. We could find the least common superclass, then make the interface an abstract class below it. Nonetheless, if the interface is implemented by classes on different inheritance hierarchies. We can find the least common superclass then create an abstract superclass below it. However, if the least common superclass is Object, it may be a good idea to leave this feature envy alone. Trying to remove

this feature envy could make the design even worse.

B.3 Calls on Objects of Library Types

Generally, in order to make a feature envy disappear, a region of code should be moved to a new home that is the class of the receiver. In other words, MOVE refactoring has to be applied. If the results from the metric implicate that the feature is more coupled to the library class, theoretically, such a feature needs to be moved. If most method calls are on an object with type defined library, there is nothing that we can do. We cannot change the library. According to feature envy metric discussed in section 5.3, the code smells detector obtains $FeatureEnvy(add, \_warehouse) = 0.851851$ which is considered a feature envy. In order to fix this feature envy, the `add` method has to be moved to the `HashMap` class. However, we cannot move this method to HashMap class, since HashMap is in `java.util` package which is not editable. Figure B.2 illustrates the discussed situation.

One way to work around this obstacle is to avoid direct use of library classes. This can be done by creating a class that extends the library class and use the newly defined class instead. This approach could help reduce the number of feature envy because we have control over such a class. Unlike dealing with library classes, there is no restriction with the user-defined class. Doing so allows us to move features between classes freely. The workaround is shown in Figure B.3.

```
public class Warehouse {
  HashMap _warehouse = new HashMap();
  void add(Item objItem, Integer objQuantity) {
    assert (objItem != null);
    if (_warehouse.containsKey(objItem))
      _warehouse.put(objItem,
                     new Integer(objQuantity.intValue() +
                                 ((Integer)_warehouse.get(objItem)).intValue()));
    else
      _warehouse.put(objItem, objQuantity);
  }
}
```

Figure B.2: Feature Envy on Objects of Library Type

```
public class WarehouseStorage extends HashMap {
  public void addItem(Item objItem, Integer objQuantity) {
    if (containsKey(objItem))
      put(objItem, new Integer(objQuantity.intValue() +
                               ((Integer)get(objItem)).intValue()));
    else
      put(objItem, objQuantity);
  }
}

public class Warehouse {
  WarehouseStorage _warehouse = new WarehouseStorage();
  void add(Item objItem, Integer objQuantity) {
    assert (objItem != null);
    _warehouse.addItem(objItem, objQuantity);
  }
}
```

Figure B.3: Indirect Usage of Library Class

B.4 May-be a Feature Envy

This case happens when we cannot really say if it is a feature envy without looking at other parts of the code. This is when analysis comes into play. Consider the code in Figure B.4. Line numbers 5, 7, and 9 are flagged for feature envy candidate because `_itemPanel` is called on three times (out of 5 total method calls). It seems to be a feature envy. However, with variable `inStock` defined on line 8, we cannot safely extract and move this portion of code to the `ItemPanel` class unless we are certain that there

```
1:  private void updateItemPanel() {
2:    Item item = getItem();
3:    int q = getQuantity();
4:    if (item == null) {
5:      _itemPanel.clear();
6:    } else {
7:      _itemPanel.setItem(item);
8:      int inStock = Warehouse.getInstance().getQuantity(item);
9:      _itemPanel.setInstock(q <= inStock && 0 < inStock);
10:    }
11:  }
```

Figure B.4: May-be a Feature Envy

are no effects on `_itemPanel` from `Warehouse.getInstance().getQuantity(item)`. Therefore, the effects analysis needs to be performed. If there are effects on `_itemPanel`, the code cannot be moved since `_itemPanel` at line 7 and `_itemPanel` at line 9 are in fact, different objects. On the other hand, if there is no write effect, it is safe to extract line 4-10 and move a newly extracted method to ItemPanel.

B.5 Exception Class

Many data classes that we found are exception classes. In fact, our earlier version of Data Class Detector reported 82 data classes in the `java.util` package, all of which are subclasses of the `Exception` class. It is reasonable that exceptions do not perform many operations. Based on this reason, the tool ignores data classes that extend `Exception`. An example of `Exception` subclass detected as a data class is shown in Figure B.5.

```
public class ProjectException() extends Exception {
  public ProjectException(String msg) {
    super(msg);
  }
}
```

Figure B.5: Exception Subclass Detected as a Data Class

B.6 Switch Statement

As discussed in Section 4.4, finding a switch statement in the code is not difficult. The real challenge how to determine which switch statement is bad. Some switch statements are legitimate and are not used in place of polymorphism.

Bibliography

[1] Giuliano Antonioil, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[2] Aqris. RefactorIt. `http://www.refactorit.com`, 2001.

[3] Cyrille Artho and Armin Biere. Applying static analysis to large-scale multithreaded Java programs. In *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75. 2001.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86. IEEE Computer Society, Washington, DC, USA, 1995.

[5] Henry G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[6] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA'05 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, San Diego, California, USA, October 16–20, *ACM SIGPLAN Notices*, 40(11):265–279, October 2005.

[7] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 151–154. ACM, New York, NY, USA, 2010.

[8] Ira D. Baxter, Andrew Yahin, Leonado Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, Bethesda, Maryland, USA, November 16–20, pages 368–377. IEEE Computer Society, Los Alamitos, California, November 1998.

[9] James M. Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *Software Engineering*, 24(2):111–124, 1998.

[10] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring improving coupling and cohesion of existing code. In *Eleventh Working Conference on Reverse Engineering*, Delft, Netherlands, November 8–November12, pages 144–151. IEEE Computer Society, November 2004.

[11] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[12] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[13] Lionel C. Briand, S. Morasca, and V. R. Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25(5):722–743, 1999.

[14] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[15] Kelvin H. T. Choi and Ewan Tempero. Dynamic measurement of polymorphism. In *Proceedings of the thirtieth Australasian conference on Computer science - Volume 62*, ACSC '07, pages 211–220. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2007.

[16] CodeSWAT. Analyst4j. `http://www.codeswat.com/cswat/index.php?option=com_content&task=view&id=43&Itemid=63`, 2012.

[17] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, London, UK, 1995.

[18] Serge Demeyer. Refactor conditionals into polymorphism: What is the performance cost of introducing virtual calls? In *Proceedings of the International Conference on Software Maintenance (ICSM '05)*, pages 627–630. IEEE Press, 2005.

[19] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, Minnesota, USA, October 15–19, *ACM SIGPLAN Notices*, 35(10):166–178, October 2000.

[20] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag, London, UK, 1999.

[21] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 404–428. Springer-Verlag, Berlin, Heidelberg, 2006.

[22] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, Oxford, UK, August 30– September3, pages 109–118. IEEE Computer Society, Los Alamitos, California, August 1999.

[23] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. *SIGPLAN Not.*, 38(11):149–168, October 2003.

[24] eclipse.org. Eclipse. `http://www.eclipse.org`, 2003.

[25] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering*, Richmond, Virginia, USA, October 28–November5, pages 97–107. IEEE Computer Society, October 2002.

[26] Twan van Enckevort. Refactoring UML models: using openarchitectureware to measure UML model quality and perform pattern matching on UML models with OCL queries. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 635–646. ACM, New York, NY, USA, 2009.

[27] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224. IEEE Computer Society Press, 1999.

[28] L. Etzkorn and H. Delugach. Towards a semantic metrics suite for object-oriented design. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on*, pages 71 –80. 2000.

[29] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021:500–517, 2001.

[30] Gern Florijn. RevJava – Design critiques and architectural conformance checking for Java software, 2002.

[31] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1037–1039. ACM, New York, NY, USA, 2011.

[32] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In *PLoP'94: Proceedings of the 1st Conference on Pattern Languages of Programs*, pages 239–257. Addison-Wesley, 1995.

[33] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, Massachussetts, USA, 1999.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachussetts, USA, 1995.

[35] Neal Glew and Jens Palsberg. Type-safe method inlining. *Science of Computer Programming*, 52(1-3):281–306, 2004.

[36] Google. Codepro analytix. `http://code.google.com/javadevtools/codepro/doc/index.html`, 2010.

[37] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of the Sixth International Conference on the Unified Modeling Language*. 2003.

[38] Daniel Graves. Incremental updating for the Fluid IR. Technical report, Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, 2004.

[39] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99: Proceedings of the 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 14–18, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer, Berlin, Heidelberg, New York, 1999.

[40] William G. Griswold and Robert W. Bowdidge. Program restructuring via design-level manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '93)*, Baltimore, Maryland, USA, pages 127–139. ACM Press, New York, May 1993.

[41] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[42] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Softw. Maint. Evol.*, 20:435–461, November 2008.

[43] Instantiations. jFactor. `http://www.instantiations.com/jfactor`, 2002.

[44] IntelliJ. IDEA. `http://www.intellij.com/idea`, 2002.

[45] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310. ACM Press, New York, NY, USA, 2000.

[46] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 264–278. Springer-Verlag, 1993.

[47] Stephen C. Johnson. Lint: a C program checker. In *Unix Programming's Manual*, pages 292–303. 1978.

[48] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transaction Software Engineering*, 28(7):654–670, 2002.

[49] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM '01)*, Florence,

Italy, November 6–10, pages 736–743. IEEE Computer Society, Los Alamitos, California, November 2001.

[50] Hyoseob Kim and Cornelia Boldyreff. Developing software metrics applicable to UML models. *Lecture Notes in Computer Science*, 2374:–, 2002.

[51] R. Kollmann and M. Gogolla. Metric-based selective representation of UML diagrams, 2002.

[52] Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. *Lecture Notes in Computer Science*, 2028:383–386, 2001.

[53] Jochen Kreimer. Adaptive detection of design flaws. In *Fifth Workshop on Language Descriptions, Tools and Applications (LDTA '05)*, pages 117–136. 2005.

[54] Jens Krinke. Identifying similar code with program dependence graphs. In *Eighth Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2–5, pages 301–309. IEEE Computer Society, October 2001.

[55] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance (ICSM '97)*, page 314. IEEE Computer Society, Washington, DC, USA, 1997.

[56] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 27–38. ACM Press, 2003.

[57] N. Maneerat and P. Muenchaisri. Bad-smell prediction from software design model using machine learning techniques. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 331 –336. may 2011.

[58] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the IEEE International Conference on Software Engineering(ICSE '03)*, pages 381–384. 2003.

[59] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Softw. Engg.*, 11:395–431, September 2006.

[60] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 133–142. IEEE Computer Society, Washington, DC, USA, 2005.

[61] Katsuhisa Maruyama and Takayuki Omori. A security-aware refactoring tool for Java programs. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 22–28. ACM, New York, NY, USA, 2011.

[62] T.J. McCabe. A complexity measure. *IEEE Transaction on Software Engineering*, 2(4):308–320, 1976.

[63] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong. Using software metrics to select refactoring for long method bad smell. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*, pages 492 –495. may 2011.

[64] Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *Proceedings of the 29th Austrasian Computer Science Conference - Volume 48*, ACSC '06. 2006.

[65] Naouel Moha. Detection and correction of design defects in object-oriented designs. In *Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 949–950. ACM, New York, NY, USA, 2007.

[66] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Lemeur. Decor: A method for the specification and detection of code and design smells. volume 36 of *IEEE Transactions on Software Engineering*, pages 20–36. 2010.

[67] Emerson Murphy-Hill and Andrew P. Black. Seven habits of a highly effective smell detector. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 36–40. ACM, New York, NY, USA, 2008.

[68] Glenford J. Myers. *Composite Structure Design*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[69] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, New York, NY, USA, 1999.

[70] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20. ACM Press, 2002.

[71] A. Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, March 1993.

[72] Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. Identifying method friendships to remove the feature envy bad smell (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 820–823. ACM, New York, NY, USA, 2011.

[73] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[74] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM conference on Computer science*, pages 66–73. ACM Press, 1993.

[75] Meilir Page-Jones. *The Practical Guide to Structured Systems Design: 2nd edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[76] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *TAPOS '97, Journal of Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[77] Linda H. Rosenberg. Applying and interpreting object oriented metrics. *Object Oriented Systems*, 1998.

[78] Emmad Saadeh, Derrick Kourie, and Andrew Boake. Fine-grain transformations to refactor UML models. In *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 45–51. ACM, New York, NY, USA, 2009.

[79] Emmad Saadeh and Derrick G. Kourie. Composite refactoring using fine-grained transformations. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '09)*, pages 22–29. ACM, New York, NY, USA, 2009.

[80] Jean-Guy Schneider, Rajesh Vasa, and Leonard Hoon. Do metrics help to identify refactoring? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 3–7. ACM, New York, NY, USA, 2010.

[81] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR '01)*, Lisbon, Portugal, March 14–16, pages 30–38. IEEE Computer Society, Los Alamitos, California, March 2001.

[82] Satwinder Singh and K. S. Kahlon. Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *SIGSOFT Softw. Eng. Notes*, 36(5):1–10, September 2011.

[83] sourceforge.net. Metrics. `http://metrics.sourceforge.net`, 1998.

[84] sourceforge.net. CppRefactory. `http://cpptool.sourceforge.net`, 2001.

[85] sourceforge.net. Transmogrify. `http://transmogrify.sourceforge.net`, 2001.

[86] sourceforge.net. JRefactory. `http://jrefactory.sourceforge.net`, 2003.

[87] Cara Stein, Letha Etzkorn, and Dawn Utley. Computing software metrics from design documents. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 146–151. ACM Press, New York, NY, USA, 2004.

[88] Eli Tilevich and Yannis Smaragdakis. Refactoring: Improving code behind the scenes. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '05)*, St. Louis, Missouri, May 15–21, pages 264–273. ACM Press, New York, May 2005.

[89] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, May 2011.

[90] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA'03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Anaheim, California, USA, October 26–30, pages 13–26. ACM Press, New York, 2003.

[91] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering(CSMR '03)*, Benevento, Italy, March 26–28, pages 91–100. IEEE Computer Society, Los Alamitos, California, March 2003.

[92] W. T. Tsai, M. A. Lopex, V. Rodriguez, and D. Volovik. An approach measuring data structure complexity. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC '86)*, pages 240–246. IEEE Computer Society, 1986.

[93] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329 –331. april 2008.

[94] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347 –367, may-june 2009.

[95] XRef-Tech. XRefactory. `http://www.xref-tech.com/speller`, 1998.

[96] Limei Yang, Hui Liu, and Zhendong Niu. Identifying fragments to be extracted from long methods. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 43–49. IEEE Computer Society, Washington, DC, USA, 2009.

[97] Yuming Zhou, Jiangtao Lu, and Hongmin Lu Baowen Xu. A comparative study of graph theory-based class cohesion measure. *Software Engineering Notes*, 29(2):13–18, 2004.

[98] Yuming Zhou, Lijie Wen, Jianmin Wang, and Yujian Chen. DRC: A dependence relationships based cohesion measure for classes. In *Tenth Asia-Pacific Software Engineering Conference*, page 215. 2003.

CURRICULUM VITAE

Kwankamol Nongpong

Place of Birth: Nakhonnayok, THAILAND

Education

B.S., Assumption University of Thailand, October 1996
Major: Computer Science

M.S., University of Wisconsin-Milwaukee, May 2000
Major: Computer Science

Dissertation Title: Integrating "Code Smells" Detection with Refactoring Tool Support

Awards and Scholarships:

2000-2004: Research Assistant, University of Wisconsin-Milwaukee.
2004-2005: Scholarship, Assumption University, Thailand.
2005-2006: Research Assistant, University of Wisconsin-Milwaukee.