ON THE AUTOMATION OF DEPENDENCY-BREAKING REFACTORINGS IN JAVA

A thesis presented in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science
at Massey University, Palmerston North,
New Zealand.

SYED MUHAMMAD ALI SHAH

2013

Abstract

Over a period of time software systems grow large and become complex due to unsystematic changes that create a high level of interconnection among software artefacts. Consequently, maintenance becomes expensive and even making small changes may require considerable resources due to change propagation in the system, a phenomenon known as *ripple effects*. Industrial evidence suggests that more resources are spent on the maintenance phase than on the initial development. It is evident that companies make huge investments to maintain legacy systems until a point comes where a complete restructuring of the system is required. In most cases, it becomes very expensive to refurbish legacy systems manually due to their inherent complexity. Several semi-automated solutions have been proposed to restructure simplified models of existing systems. It is still expensive, in terms of resources, to translate those model level transformations into source code transformations or refactorings. The question that arises here is whether we can automate the application of model level changes on the source code of programs.

In this thesis, we have developed novel algorithms to automate the application of a class of architectural transformations related to improving modularity of existing programs. In order to evaluate our approach, we have analysed a large dataset of open source programs to determine whether the manipulation of models can be translated into source code refactorings, whether we can define constraints on those refactorings to preserve program correctness, and to which extent the automation of the whole process is possible. The results indicate that this automation process can be achieved to a significant level, which implies that certain economic benefits can be gained from the process.

Acknowledgements

I would like to express my profound gratitude to my supervisor Dr. Jens Dietrich for his advice, guidance, and endless support through every step of the way. He has a great passion for his work and he knows how to get the best out of his students. He has been a source of learning throughout these years. I also thank him for the Guery tool he had developed. This tool was very helpful in my research.

I extend my sincere gratitude to my co-supervisor Dr. Catherine McCartin for her valuable feedback during the entire time. She has been very kind and helped me polish my work. This work would have not been possible without the help and support of my supervisors.

A special thanks to my family. Words cannot express how grateful I am to my mother (late), and father for all of the sacrifices that they have made on my behalf. Their prayer for me was what sustained me thus far. Thanks to all of my siblings for their support and wishes during these years. I would like to thank my sisters Farhat, Yasmin, and Najma for looking after me so well, whenever I visited home. A word of thanks to my younger brother Taskeen, who helped me in many different ways during my research.

My years at Massey were very enjoyable, thanks for the friendship of Abrar (for all the enjoyable distractions), Shujat (for the jokes and laughs), Ezanee (for stimulating useful discussions), Saleem (for being a good flatmate and friend), and Tariq (for the friendship and discussions on every aspect of life). I would also like to thank my friends in Pakistan especially Irfan, Bhatti, Ainan, and Naseer. Many thanks to the Pakistani community in New Zealand. They have been very kind and never made me miss the exotic Pakistani cuisine. They all made me feel at home.

I take this opportunity to thank Massey University and the School of Engineering and Advanced Technology (SEAT) for providing a conducive working environment for research. I would also like to thank the SEAT staff, in particular, Christina Bond, Fiona, Michelle Wagner, Linda Lowe, and Dilantha Punchihewa for never complicating the administrative tasks.

Last but not least, I would like to thank the Higher Education Commission for providing the financial support granted through Overseas Scholarship Program to complete my PhD degree.

This thesis is dedicated to my late mum ...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Definition

Software is prevalent in every sphere of life, whether it is government, banking, health-care, communication, transportation, or retail sectors, the use of software is increasing by the day. In 2010, the volume of the global software industry was 265 billion US dollars (an increase of 7.1% from the previous year) and it is forecast to reach an estimated 357 billion US dollars by 2015 (Research and Markets, 2011). These figures indicate that the software industry will continue to grow over the coming years.

According to Lehman, software that is used in the real-world environment must change or it becomes ineffective progressively (Lehman, 1979). We notice changes happening around us on regular basis, including new government regulations, tax law modifications, growth in businesses, and manufacturing of new hardware. These changes must be reflected in our software too. In addition, faults are revealed in software after its deployment, which leads to changes in order to fix those faults, also known as *bug fixes*. These factors are the driving forces that put software under continued change pressure during its entire life cycle.

The increasing role of software in our everyday lives and businesses puts constant pressure on software developers to perform. They have to deliver new applications or new releases of existing applications on strict deadlines. Compounding this, development teams are often outsourced or distributed geographically. This can result in uncoordinated work that leads to unsystematic changes and workarounds, creating a high level of interconnection among software artefacts, such as models, subsystems, and libraries. This interconnectivity often increases to a point where it becomes difficult to understand a part of the system without understanding other parts, introducing *accidental complexity*[1]. The situation gets worse when new developers join teams. It becomes harder for them to understand the existing system to make changes and eventually when more changes are introduced, the quality declines even further. Therefore, such systems end up in an accelerating loop where the quality declines over a period of time and it becomes very difficult and expensive to maintain them, as a small change may propagate to other parts of the system. Ultimately, some systems may turn into a *Big Ball of Mud*, a term which is defined by Foote and Yoder (1997) as follows:

> *"A big ball of mud is haphazardly structured, sprawling, sloppy, DuctTape and bailing wire, SpaghettiCode jungle. ... These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition."*

The software maintenance phase constitutes a significant part of the software development life cycle. This includes adding new features, improving performance, fixing problems, and migrating to a new technology or platform. Industrial evidence shows that companies invest more resources on maintaining and evolving their software than on the initial development (Coleman et al., 1994; Schach, 1996). According to Schach (1996), 67% of total software development cost is spent on the maintenance phase. The author calculated the maintenance cost from various sources by averaging data from multiple projects between 1976 and 1981. Furthermore, in 1992 an analysis of 132 projects at Hewlett-Packard revealed that 40 to 60 percent of the total cost was spent on maintenance (Grady, 1994). Some organisations even spent 80% of their resources on maintaining their software (Yourdon, 1992). This data indicates that maintenance is an expensive and extremely time-consuming phase in the software life cycle.

The question that arises is how to reduce maintenance costs and improve maintainability of software so that it is easier to modify and evolve over its life cycle. One of the solutions to improve software maintainability is to introduce modularity into the system. Software modularity is an important principle in software engineering for building large and complex systems[2]. It refers to a logical decomposition of software design where a complex system is divided into independent modules so they become manageable and maintainable (Parnas, 1972). In a modular system, every module

[1]Accidental complexity as opposed to essential complexity creeps into a system through unsystematic changes and workarounds making the system harder to understand (Brooks, 1987).

[2]By complex here we mean essential complexity as defined by Brooks (1987), because software reflects systems that are complex in nature, such as flight control systems.

works relatively independently from other modules. Therefore, when changes are required in a module, there is no need for the developer to understand the entire system. The individual modules can be tested and deployed separately. The loose coupling between modules is the key to reducing maintenance costs.

According to Knoernschild (2012), there are two facets of modularity. He calls those the design time model and the runtime model. The design time model refers to efforts made to design the software architecture in a modular way. This includes applying design principles and patterns, decomposing modules at the right level of abstraction, maintaining low coupling and high cohesion, managing dependencies and so on. The advantage of design time modularity is that modules can be evolved and tested separately. In addition, development teams can also work independently on different parts of a project and once the development is complete, a build tool (maven, ant, make, ivy) can be used to produce the final executable product such as a Java jar file (Blewitt, 2009). However, it is important to correctly bundle different parts of a system into their respective modules (jars). In this way, we not only achieve design time modularity but also runtime modularity.

Runtime modularity, as opposed to design time modularity, focuses on managing modular systems at runtime. There are several benefits of runtime modularity. This includes the ability of an application to dynamically deploy modules, deploy different versions of modules together, resolve dependencies dynamically, and swap components at runtime without restarting the application. Runtime modularity can be supported by modular platforms, such as OSGi. We are interested in this kind of modularity where the focus is on deployment rather than on development of software. This means we aim to transform monolithic applications[3] into modular architectures so they can achieve the aforementioned benefits of runtime modularity. From this point, by modularity we mean runtime modularity.

In recent years several techniques have been introduced to facilitate software modularisation (Johnson et al., 2005; Walls, 2009). In order to take advantage of modern modularisation techniques many software vendors are refactoring their existing monolithic products to modular architectures. For instance, the Jigsaw project has been initiated to refactor the Java Development Kit (JDK) into a modular architecture (Project Jigsaw, 2008). One of the main objectives of this project is to introduce a module system in the JDK, which would improve certain metrics such as download time, loading time and memory footprints.

It is interesting to look into the current state of the JDK to be able to understand the need for its modularisation. The JDK has grown extremely large since its initial release

[3]Applications that are built around a single core and are difficult to change due to change propagation.

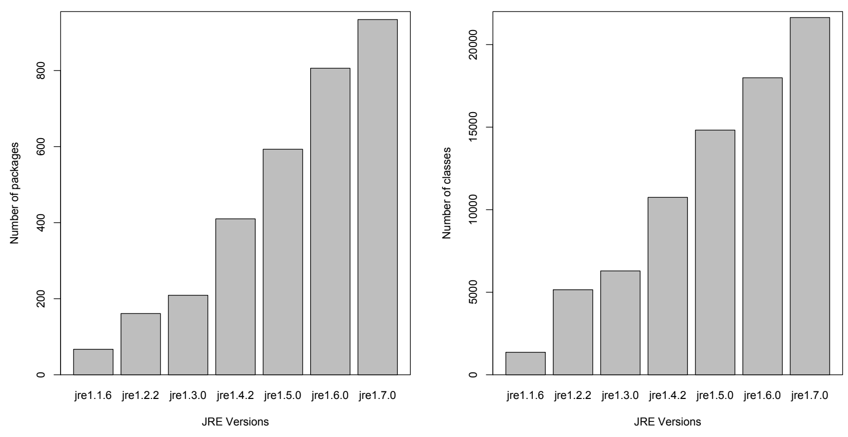


Figure 1.1: Evolution of Packages and Classes in JRE

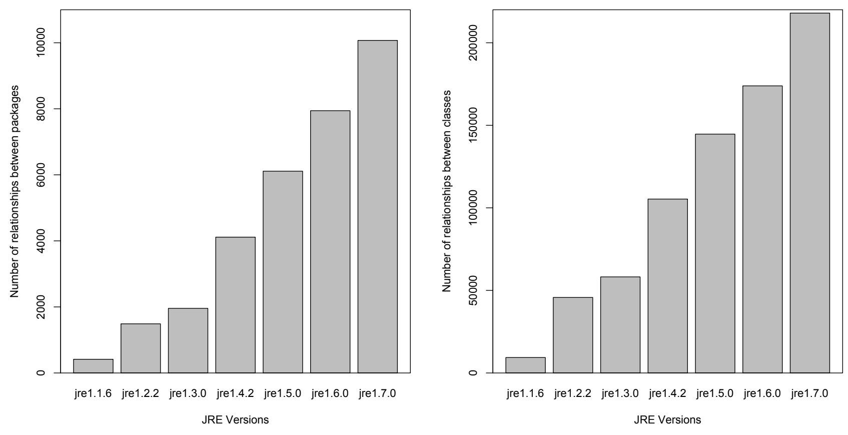


Figure 1.2: Evolution of Package Relationships and Class Relationships in JRE

and many classes and packages have been added to the standard library (Figure 1.1, 1.2). It is not only a large number of classes and relationships that make the JDK complex, but also the presence of a large number of class and package level tangles. A tangle is a strongly connected component in a graph where every node is either directly or indirectly linked to every other node. According to established design heuristics, the presence of tangles is a sign of degraded design (Parnas, 1972; Stevens et al., 1979). The evolution of tangles and their relationships is shown in figures 1.3 and 1.4 respectively. The incremental increase in the number of tangles indicates how the structure of a system erodes over the period of time.

As an example, let's examine the structure of the JRE 1.7.0. Figure 1.5 shows the

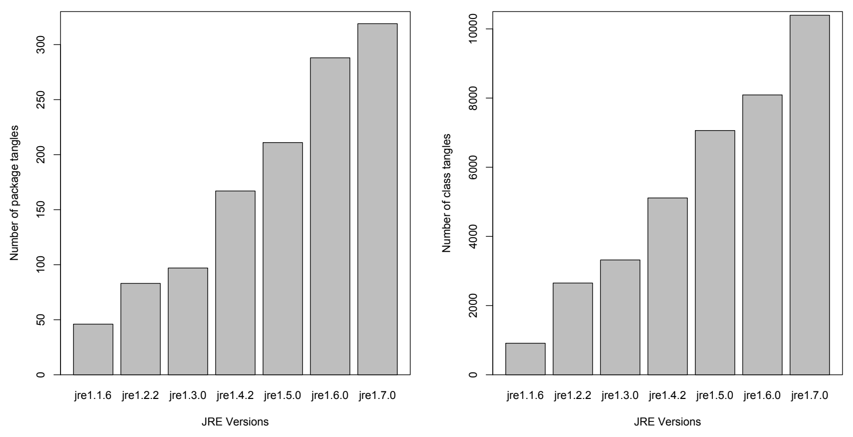


Figure 1.3: Evolution of Package and Class level Tangles in JRE

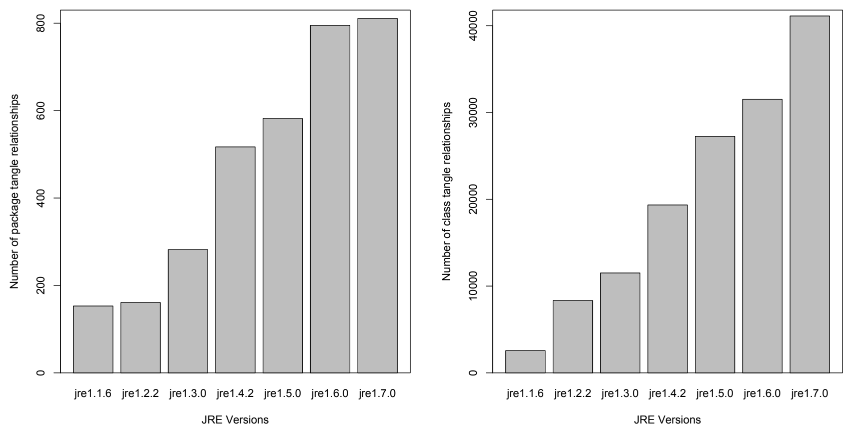


Figure 1.4: Evolution of Relationships of Package and Class level Tangles in JRE

dependency graph[4] of the JRE 1.7.0 packages (934) and their relationships (10072), while figure 1.6 shows the dependency graph[5] of classes (21638) and their relationships (217973). JRE 1.7.0 has 10394 class level tangles with 41123 relationships between tangles. In a similar way, there are 319 package level tangles with 811 relationships. It is clear from these figures that the JRE has a complex architecture and it is a classic example of what is known as a *Big Ball of Mud*. It turns out that this is a common issue. A study by Melton and Tempero (2007b) has revealed that many real-world programs have similar issues.

[4]Source: Massey Architecture Explorer `http://goo.gl/hc38k`
[5]Source: Massey Architecture Explorer `http://goo.gl/hwkhT`

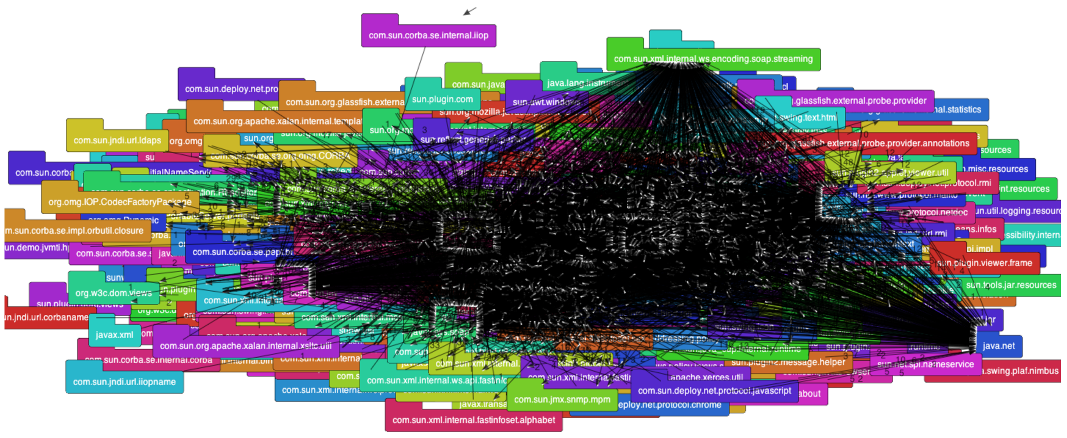


Figure 1.5: JRE 1.7.0 Package Level Dependency Graph

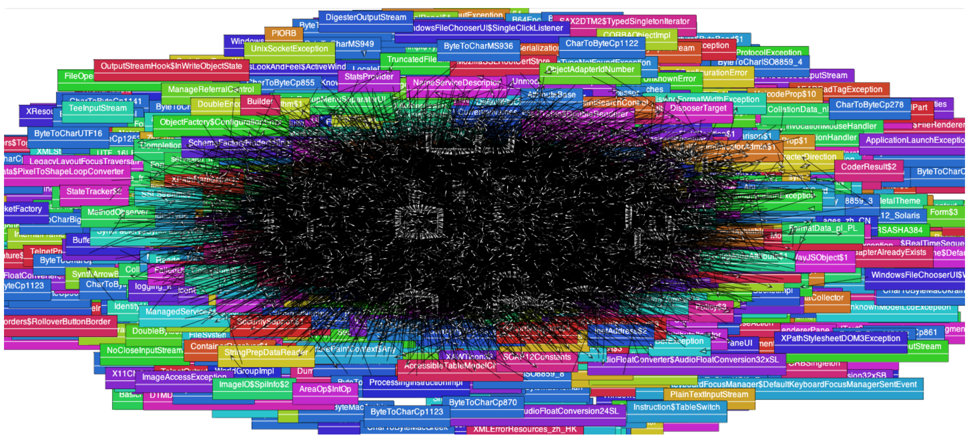


Figure 1.6: JRE 1.7.0 Class Level Dependency Graph

The project jigsaw started in August 2008 and is still under development due to the complex monolithic architecture of the JDK. The JDK was built around a monolithic architecture, therefore dependencies between different software artefacts became so deep rooted that the modularisation process has turned out to be a challenging task. This project was scheduled to complete with the release of Java SE 8, but now it has been rescheduled until the release of Java 9. According to Mark Reinhold, the chief architect of the Java platform group, one of the reasons for delay in the project is the deep interconnectivity at the API and implementation levels of the JDK. This makes it difficult to remove several dependencies that compromise the modularisation process (Reinhold, 2012). This indicates that refactoring of complex legacy systems is a resource demanding task and that solutions should be proposed to automate the whole process.

In this thesis, our approach is based on the assumption that modularity manifests itself by the presence (or absence) of architectural antipatterns such as circular dependencies between modules (Dietrich et al., 2010). The modularity of a system can be measured by analysing a simplified architectural model of the system and by counting the presence of architectural antipatterns. Empirical studies (Melton and Tempero, 2006; Dietrich et al., 2010) have shown that real-world systems are ripe with these kinds of architectural antipatterns, indicating a lack of modularity.

For example, in OpenJDK 1.6.0 b-14, there is cyclic dependency between AWT and Swing: `java.awt.Component` uses `javax.swing.JComponent`[6] and `javax.swing.JComponent` extends `java.awt.Container`. The dependency from `JComponent` to `Container` is necessary because this allows Swing to use some general UI functionality defined in AWT. However, the reference from the older AWT toolkit to the newer Swing toolkit points to a serious problem. This means an application that is developed using AWT cannot be deployed without the resource-demanding Swing toolkit. AWT components perform better because they use native APIs such as Direct X on Windows whereas Swing components handle their own rendering, making them resource demanding.

Moreover, It is interesting to see that the dependency from `java.awt.Component` to `javax.swing.JComponent` is not present in the Apache Harmony (Harmony, 2010) implementation of the JDK version 6.0, r917296-snapshot. This means that it is possible to break dependency edges between these classes without compromising the external behaviour of the system. We refer to such *model refactoring*[7] as *dependency-breaking refactoring*. The question arises how these critical dependencies can be found and removed systematically.

1.2 Research Questions

In this thesis, we aim to develop novel algorithms that can provide an automated solution to apply model level refactorings on the source code of programs. That is, given a set of critical dependencies we remove or reorganise those dependencies on the source code of programs. This leads to the first question:

1. Can model level dependency-breaking refactorings be automatically translated into source code refactorings?

[6]Source: Massey Architecture Explorer: `http://goo.gl/0U8Gz`

[7]According to Fowler (1999), refactoring is an activity for improving the internal software structure without affecting its external behaviour. It helps in achieving quality attributes, such as modularity, maintainability and evolvability of a software system. Model refactoring refers to transformations performed on software models, such as dependency graphs, UML diagrams etc.

Refactorings introduce changes in the existing system, which is a risky and expensive task in the sense that it may break the system. In order to mitigate the risks associated with the change and to ensure program correctness, we need a formalism to define the correctness of a refactoring and tools that can verify refactorings against this. In this thesis, we aim to develop appropriate constraints for refactorings that preserve program correctness. This defines our second research question:

2. How can we define and evaluate constraints on code level refactorings to preserve the correctness of the program being refactored?

It is possible that not all refactorings on the model level succeed on the source code level. Therefore, we try to quantify this based on a dataset. This defines our last question:

3. To what extent can model-to-code refactorings be automated?

1.3 Approach

Our approach is to develop novel algorithms and techniques to remove critical dependencies from existing programs. We have focused on critical dependencies that undermine the modularity of programs. These critical dependencies can be identified from architectural antipatterns that compromise modularity (Dietrich et al., 2010). Our proposed methodology is to analyse architectural antipatterns on the model level, to detect critical dependencies between software artefacts that participate in a large number of antipattern instances, and to safely remove these critical dependencies from the source code of programs.

1.3.1 Critical Dependency Detection

In this thesis, we have attempted to remove architectural antipatterns that are widely accepted as designs problems (discussed in Section 2.3). These antipatterns are known to compromise modularisation of programs. Hence, our approach is based on the assumption that the removal of the chosen antipatterns will improve the modularity of programs.

1.3.1.1 Metrics

We have used fitness functions in terms of metrics to ensure that our refactoring process has achieved the intended improvements. We have analysed several metrics for this purpose, which are discussed in section 4.5.

1.3.2 Tools

In order to implement our algorithms, we have developed an Eclipse plugin named CARE (Code and Architectural Refactoring Environment). The purpose of this plugin is to identify and execute refactorings on the source code level with a push of a button. The plugin has the ability to analyse a single program or batch-script multiple programs in the Eclipse workspace environment. We have chosen the Eclipse platform because it provides a powerful Abstract Syntax Tree (AST) API, which can be used to manipulate the source code of a program. This API is well documented and has a good active community. This plugin uses the Guery (Dietrich and McCartin, 2012) library for the analysis of architectural antipatterns on the model level.

The architectural model we have used is a dependency graph consisting of classes[8] and their relationships (Mancoridis et al., 1998). The dependency graph can be extracted from the bytecode or source code of a program (discussed in Section 2.1). In the dependency graph classes are represented as nodes, while edges represent relationships between classes. Dependency graphs provide an abstract representation of (the design of) a program, which is suitable in detecting architectural antipatterns.

1.3.3 The Dataset

In recent years corpora-based empirical investigations have become common due to the availability of open source software (Melton and Tempero, 2006; Tempero, 2008; Dietrich et al., 2010; Taube-Schock et al., 2011). The use of a corpus makes it convenient to benchmark different techniques against a set of data that is available to all researchers. In this thesis, we have investigated research questions through an empirical study on a large set of open source programs written in the Java language. Our dataset contains a large variety of real-world programs which gives us the opportunity to apply our techniques on a broad spectrum of programs (see further discussion in Section 2.6).

[8]We use the term classes to represent non-array Java reference types. In particular it includes interfaces and enumerations

1.4 Thesis Contribution

1.4.1 Algorithms

The first contribution of this thesis lies in the development of novel algorithms that automatically translate model level refactorings to source code refactorings. In particular, we apply our algorithms to solve software dependency problems related to modularisation. Our approach aims to provide a complete solution, that is, we detect architectural problems on the model level, identify appropriate refactorings and apply them on the source code. Existing solutions focus mainly on the detection of architectural problems. Our second contribution lies in the selection of refactoring constraints. We aim for constraints that provide optimum results, that is, constraints that are fast to evaluate, easy to implement and have a high success rate.

1.4.2 Implementation

We have developed an Eclipse plugin *CARE* to implement the algorithms. This plugin makes use of standard Eclipse refactorings to perform the desired actions. Additional refactorings have been implemented as required (see discussion in Section 5.4.3). The plugin can identify and execute potentially high-impact refactorings on the source code of programs. This tool is available online and can be downloaded from the project website[9].

1.4.3 Validation

In order to evaluate our approach, we have applied it to a large set of real-world programs, which are part of the Qualitas Corpus (Tempero et al., 2010). The Qualitas Corpus is a collection of open-source Java programs widely used in empirical studies (discussed in Section 2.6). We have performed several experiments to answer the research questions posed in section 1.2. We have used the CARE plugin to perform experiments and find appropriate refactorings to remove architectural antipatterns. Our study is unique as it is validated through an experiment on a large, independent dataset.

[9]`http://code.google.com/p/care/wiki/Documentation`

1.5 Thesis Structure and Outline

The rest of the thesis is structured as follows:

Chapter 2. This chapter provides an overview of the research methodology. Here we discuss the architectural model and architectural antipatterns used for the analysis. The mechanism for detecting critical dependencies in programs is described. The chapter also includes discussion about different dependency classifications and the design of our experimental setup.

Chapter 3. This chapter investigates different refactoring techniques used to break dependencies between classes. The chapter concludes with the selection of appropriate refactorings for decoupling classes.

Chapter 4. This chapter describes our work on package level refactorings, in particular the "move class" refactoring. Several metrics are defined in this chapter, which are used as fitness functions. These metrics are used to asses the quality of programs after refactorings. The chapter explains the algorithm used to break dependencies between classes and packages. Pre and postconditions of refactorings are also discussed. The results of our experiment are discussed, along with the implementation details.

Chapter 5. This chapter presents our work on a combination of package and class level refactorings. An algorithm for composite refactorings is also presented. Here we discuss pre and postconditions related to class level refactorings. Finally, this chapter concludes with a discussion about results obtained from experiments.

Chapter 6. This chapter brings to conclusion the research presented in this thesis. Here we discuss threats to validity, research contribution, and directions for future research.

Chapter 2

Research Methodology

In this chapter, we describe the methodology used to answer the research questions developed in chapter 1. This chapter begins with an overview of the architectural model used to perform the analysis of programs. This is followed by a detailed description of architectural antipatterns. Here we also explain the selection process for refactorings and describe tools used in experiments. These tools provide the ability to detect refactoring opportunities and apply appropriate refactorings on the actual programs. Finally, the dataset and the experimental setup are described.

2.1 Architectural Model - The Dependency Graph

A model is a simplified representation of a real system, which abstracts from unnecessary details and provides an overall picture of the system. According to Lethbridge and Laganiere (2002), an architectural model is a simplified representation of software architecture for the sake of expressing a viewpoint. For instance, an architectural model can express a view of the logical decomposition of subsystems or modules of the system. Its main purpose is to better understand the system and to be able to communicate the architecture with other stakeholders that are either directly or indirectly related to the software system.

The architectural model we have used is a program dependency graph (a directed graph) of classes and their relationships (Mancoridis et al., 1998). Dependency graphs provide an abstract representation of a program. The idea of using the dependency graph as a representation of the software architecture is not new. This has been widely used in software architecture analysis tools such as ReStructure101 (2008), Sotograph (Bischofberger et al., 2004), Restructure101 (ReStructure101, 2008), JDepend (Clark, 2003), and Guery (Dietrich and McCartin, 2012).

2.1.1 Extracting the Model

In the dependency graph, each reference type in a program (class, interface, enum etc) is represented as a vertex (node). Additional properties such as namespace (package name), visibility and abstractness are represented as labels on vertices. Edges represent relationships between classes and are labelled with relationship types. The three supported relationship types are *uses*, *extends*, and *implements*. The relationship types *extends* and *implements* represent inheritance, while the relationship type *uses* represents all other types of references between classes, such as type references in methods and fields.

The dependency graph can be extracted from the bytecode or source code of a program. Dependency graphs that are extracted from bytecode are slightly different from source code, as references to constant primitive values declared as *static* and *final* are replaced with their respective values by the Java compiler. This is also known as constant folding (Muchnick, 1997). In our experiments, we have used bytecode to generate dependency graphs. The reason for choosing bytecode over source code for generating dependency graphs is mainly due to the availability of robust bytecode analysis tools. In addition, the use of bytecode to build dependency graphs gives a wider scope to potentially using other modern languages such as Scala, JRuby and Jython that compile to Java bytecode[1] and rely on the JVM as an execution platform (Da Vinci Machine, 2008). We have used the Dependency Finder (Tessier, 2010) to extract the dependency graph from bytecode of existing programs. This API has certain limitations. In particular, it does not support generic types while building the dependency graph from bytecode. We do not consider this as a problem since we are more interested in dependencies at runtime rather than during compile time[2]. We have used the Java Universal Network Graph (JUNG) API (Madadhain et al., 2005) to represent dependency graphs.

2.2 Architectural Antipatterns

During the code review phase, experienced developers often note weaknesses in code and structure of a system. These weaknesses are commonly referred to as smells in the system; as they point to potential design problems. The term "code smell", introduced by Fowler (1999), indicates symptoms of a deeper problem in the source code of a

[1]In our experiments, we have used programs written in Java language only.
[2]Generic type information is erased during compile time through a process called *type erasure*.

program, for example, duplicate code, long methods, large classes etc. In his book on refactorings, Fowler (1999) identifies several code smells and provides a catalogue of refactorings that can be used to deal with these code smells. Demeyer et al. (2009) use code smells as an indicator to recognise that a legacy system needs to be re-engineered. A closely related area to code smells is antipatterns. According to Brown et al. (1998), "An antipattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences". The concept of antipatterns is inspired by Gang of Four's book on design patterns (Gamma et al., 1994), which are a catalogue of good design whereas antipatterns represent a catalogue of bad design. Antipatterns can exist at code, design, and architecture levels (Koenig, 1998).

There exists another class of smells that indicates problems in higher levels of abstractions, such as cycles between namespaces. Lippert and Roock (2006) refer to these smells as architecture smells, further discussed by Stal (2008). A code smell, for instance, may point to a bad reference between individual classes, whereas an architecture smell may indicate bad coupling between modules or subsystems. We are interested in this class of smells and refer to these smells as architectural antipatterns, since they represent design flaws that negatively affect the quality of the overall system.

2.2.1 Antipattern Detection Tools

There exist several tools, widely used in the software engineering community, to detect code smells such as PMD (Copeland, 2005), Checkstyle(Burn, 2008), and Findbugs (Hovemeyer and Pugh, 2004). These tools identify code smells within compilation units of a Java program. On the other hand, a number of architectural analysis tools have been developed to identify architecture smells across a whole program. These tools include Sotograph (Bischofberger et al., 2004), Lattix (Sangal et al., 2005), ReStructure101 (ReStructure101, 2008), Crocopat (Beyer and Lewerentz, 2003), JDepend (Clark, 2003), Massey Architecture Explorer (MAE) (Dietrich, 2012), and Guery (Dietrich and McCartin, 2012).

2.2.2 Evaluation of Tools

Requirements

In order to choose the right tool for the analysis of programs, we laid out the following requirements:

1. The tool should have built-in declarative language support in order to define custom architectural antipatterns.
2. It should provide scripting support to analyse a large set of programs.
3. It should be able to compute transitive closure of relationships in order to represent indirect dependencies.
4. Moreover, it should be scalable to allow analysis of large programs, as our dataset contains relatively big programs.
5. Finally, it should provide insight into architectural metrics to asses the quality of architecture.

Tool	Script-ability	Custom Queries	Transitive Closure	Architecture Metrics	Scalable
Guery-1.4.0	YES	YES	YES	YES	YES
CrocoPat-2.1.4	YES	YES	YES	NO	NO
JDepend-2.9.1	YES	NO	NO	YES	N/A
Lattix-7.8	NO	NO	NO	YES	N/A
ReStructure101	NO	NO	NO	YES	N/A
Sotograph	NO	NO	NO	YES	N/A

Table 2.1: Tool Features in terms of Architectural Antipatterns Detection

Final Selection

Table 2.1 shows a comparison of different tools based on requirements. On the basis of our requirements and evaluation of several tools, the tool we have chosen to detect architectural antipatterns instances is Guery (Dietrich and McCartin, 2012). This tool has an easy-to-use query language and a scalable implementation of the query engine that can detect antipattern instances in large graphs represented using JUNG library (Madadhain et al., 2005). Guery provides scripting support to analyse a large set of dependency graphs of programs. We conducted some experiments and found that this tool scales better than Crocopat (Dietrich et al., 2010). Scalability tests were performed on the basis of custom queries (antipattern definitions) that were executed against dependency graphs of programs. We did not perform scalability tests on tools such as JDepend, Lattix, ReStructure101, and Sotograph because these tools do not provide support to define custom architectural antipatterns. This has made it impossible to compare performance. The commercial architecture analysis tools including Lattix, ReStructure101, and Sotograph are mainly UI centric, which makes them unsuitable for scripting the analysis of a large set of programs.

2.2.3 Representing Antipatterns

Visual Notation

We use visual definitions to illustrate architectural antipatterns. Boxes inside packages represent classes and arrows represent paths (sequence of edges). Paths can traverse more than one package. Cardinality constraints such as (1:M) represent the minimum and the maximum path length, where "M" is unbound. The constraint (1:1) means that this path has a single edge, while (0:M) means there could be a path of length zero (an empty path). Paths are labelled with either *uses* relationship or inheritance relationships (*extends* or *implements*).

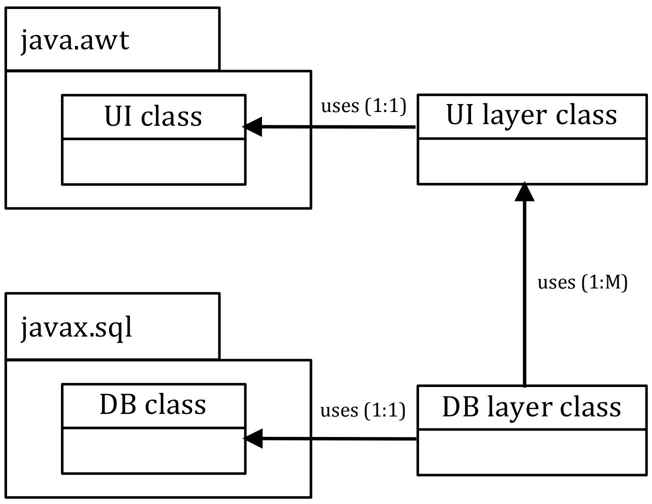


Figure 2.1: User Interface/Database Dependency Antipattern

Figure 2.1 shows a visual definition of an antipattern representing a layer violation. This antipattern represents a well known design problem where a database layer depends on a user interface layer, violating the 3-tier architecture principle. In the presence of this antipattern it becomes difficult to replace the user interface layer. In this example, layer membership is formalised as a direct reference to UI or DB classes respectively, which exists in standard libararies such as AWT (for UI classes) or JDBC (for DB classes). There should be a non-empty path connecting the DB layer class with the UI layer class.

```
1  // no class should depend on UI and DB classes
2  //(and therefore the respective layers) at the same time
3  motif db2ui
4  select uiclass, uilayerclass, dbclass, dblayerclass
5  where "uiclass.namespace=='java.awt'
6     and "dbclass.namespace=='java.sql'"
7  connected by db2ui(dblayerclass>uilayerclass)[1,*]
8     and dblink(dblayerclass>dbclass)[1,1]
9     and uilink(uilayerclass>uiclass)[1,1]
10 group by dblayerclass.namespace, uilayerclass.namespace
```

Listing 2.1: User Interface Dependencies in Persistent Layer

Graph Queries

In Guery, a domain specific language is used to define an antipattern as a graph query to be run on the dependency graph of a program. The Guery syntax borrows elements from SQL and object-oriented expression languages. An antipattern corresponds to the concept of a *motif* in network theory. The motif instance detection in the dependency graph of large programs is a non-trivial task. It is an NP-complete problem (subgraph isomorphism problem). The Guery solver uses several design heuristics and dynamic programming techniques to achieve the best possible results (Dietrich et al., 2012).

Listing 2.1 shows the definition of an antipattern representing a layer violation. We define this query in a directed graph of classes (vertices) and their relationships (edges). Additional information, such as the namespace of a class, is represented as labels on vertices. In the definition, single line comments (line 1,2) and naming of the antipattern (line 3) are supported. The *select* keyword (line 4) binds vertex roles (like uiclass, dbclass) to vertices in the graph. The *where* clause (line 5) defines a constraint on the vertex roles, which is a boolean expression referring to a vertex. The constraints are defined using string literals that are interpreted by MVEL (MVFLEX Expression Language) (MVEL, 2009). The MVEL expression engine compiles these constraints into bytecode for faster evaluation. Path roles (like dblink, uilink) represent sequences of edges and constraints can also be defined on them. The keyword *connected by* is used to define path roles. A path can also be constrained by cardinality constraints, such as [1,*] meaning the minimum number of edges is 1 and the maximum number is unbound (*). Similarly a constraint [0,*] means a path may be empty, that is, it has zero edges. A cardinality constraint of [1,1] means that the path will have only a single edge. The *group by* keyword is used to combine results based on a particular class or namespace.

2.3 Antipattern Set

2.3.1 Overview

In this thesis, we have chosen architectural antipatterns based on the principles of object-oriented design and by focusing on two meta-patterns related to modern modular platforms. Object-oriented design principles provide guidelines to be followed to avoid "bad" design - design which is difficult to modify and expensive to maintain. According to Martin (2003), there are several characteristics of bad design that should be avoided:

- System Rigidity: Very hard to change because changing one part requires changing other parts of a system, also known as *ripple effect*.
- System Fragility: Making a change breaks unrelated parts of the system.
- Code Immobility: Difficult to reuse because it is hard to untangle from the current system.

In order to avoid bad design, there exist a number of design principles. For example, dependency inversion principle, acyclic dependency principle, stable abstractions principle[3] etc. In the next sections, we describe these design principles along with their respective antipattern, which violates the design principle.

On another level, we are interested in a certain class of architectural antipatterns that compromise modularisation. In particular, those antipatterns that hinder transforming monolithic architectures to modular architectures. We are interested in this particular class because many vendors are migrating their monolithic products to modern platforms. For instance, recently leading commercial application servers Weblogic and Websphere were modularised and refactored to adopt OSGi (Walls, 2009) platform. The modularisation of the Java Development Kit is also in progress as discussed in section 1.1 (Project Jigsaw, 2008). This clearly shows the trend towards modularization.

At the core of OSGi and related dynamic component platforms, there exist two meta-patterns: namespace separability and interface separability (Dietrich et al., 2010). The term *namespace separability* refers to how easy it is to separate namespaces from one another. In modern modular platforms, it is important for an application to have loose coupling between packages (namespaces) so that they may be separated in different modules (bundles). As opposed to standard Java, packages play an important role in the OSGi environment. In OSGi and similar frameworks, additional semantics are

[3]`http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign`

added to packages to control their visibility. These additional semantics contribute to software encapsulation, which leads to decoupling of the life cycle of components and their associated packages. The benefits of this decoupling include swapping of components providing the same service at runtime, dynamic deployment of modules, and parallel system upgrades.

The term *interface separability* refers to how easy it is to separate interfaces from their implementations. The benefit of this includes swapping service implementations at runtime without rebooting a system. Service implementations are rewired during runtime through dependency injection frameworks (Fowler, 2001). This mechanism is available in all dynamic component models, such as OSGi and its extensions, and Spring Dynamic modules. In order to take advantage of these modern platforms, dependencies between interfaces and their implementations should be avoided, and this also violates the dependency inversion principle.

The reason for choosing a particular set of antipatterns is because we are interested in refactoring monolithic applications into modular architectures. We are also interested in finding out what hinders in achieving that (Dietrich et al., 2010). As mentioned before, there are two essential things that should be done in order to transform monolithic applications into module systems, i.e., separate namespaces and separate interfaces from their implementations. The antipattern set that we have chosen compromises the separability property. There exist other antipatterns (Lippert and Roock, 2006; Stal, 2008) that compromise the quality of architecture, but we have limited the scope by focusing on a set of four antipatterns that compromise modularity in modern modularisation platforms. However, the implementation of our tool allows adding new project specific antipatterns or any other antipatterns without changing the source code of the tool. These antipatterns can be formalised as graph queries that run against graphs to detect antipattern instances. Here is the list of these antipatterns:

1. Circular Dependencies between Packages (CD) - affects namespace separability
2. Subtype Knowledge (STK) - affects interface separability
3. Abstraction Without Decoupling (AWD) - affects interface separability
4. Degenerated Inheritance (DEGINH) - affects interface separability

2.3.2 Circular Dependencies between Packages

In Java, packages are mainly used as namespaces to group classes with related functionality. According to Martin (2000), classes should be grouped in a way which facilitates an independent build, test and release cycle. He defines the Acyclic Dependency Principle (ADP), which states that packages should not be involved in circular dependencies. The dependency cycle between packages is a variation of "cycle between modules" discussed by Stevens et al. (1979). The circular dependency between packages is regarded as a design flaw, which implies that packages involved in cycles cannot be deployed and evolved independently. Such a situation arises when classes in a package access classes in another package and vice versa. Packages that are involved in circular dependencies become less maintainable and are difficult to reuse (Abdeen et al., 2010). Empirical studies have shown that circular dependencies between classes and packages are very common in open source Java programs (Melton and Tempero, 2006; Dietrich et al., 2010).

Example

In OpenJDK 1.6.0 b-14, there is cyclic dependency between AWT and Swing: `java.awt.Component` uses `javax.swing.JComponent` and `javax.swing.JComponent` extends `java.awt.Container`. This dependency is critical because applications developed using AWT cannot be deployed without the Swing toolkit. Listing 2.2 shows the source code of one dependency, which creates this CD instance and figure 2.2 shows a visual diagram of the dependency path.

```java
package java.awt;
public abstract class Component ... {
  ...
  if (Component.isInstanceOf(this, "javax.swing.JComponent")) {
    if (((javax.swing.JComponent) this).isOpaque()) {
      ...
    }
  } ...
}
```

Listing 2.2: Reference from class Component to class JComponent in JDK 1.6.0 b-14

Definition

We have used a strong definition of circular dependency (SCD) where there is a single path that creates a cycle between packages, as shown in figure 2.3. This definition is different from the standard definition used in many places in the literature (Stevens et al., 1979; Beyer and Lewerentz, 2003; ReStructure101, 2008) where there might not

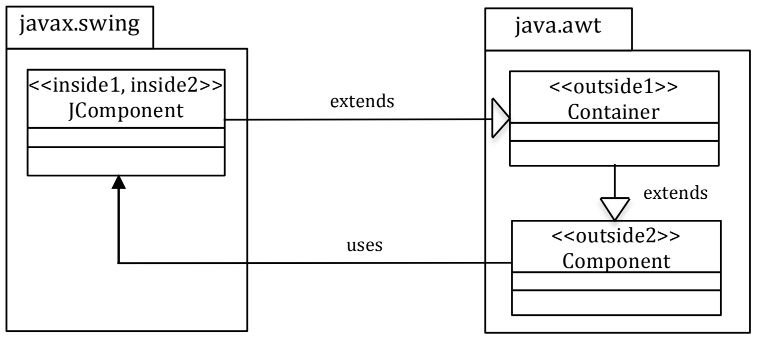


Figure 2.2: Dependency Cycle Between the AWT and Swing Packages

be a single dependency path that creates a dependency cycle between packages. We refer to this kind as weak circular dependency, as shown in figure 2.4. On the other hand, our SCD antipattern represents a strong coupling between packages and cannot be easily refactored by splitting packages, whereas a weak circular dependency can be removed by simply splitting packages. For example, in figure 2.4 if *package2* is split into two packages with classes C and D in different packages, we can get rid of the dependency cycle.

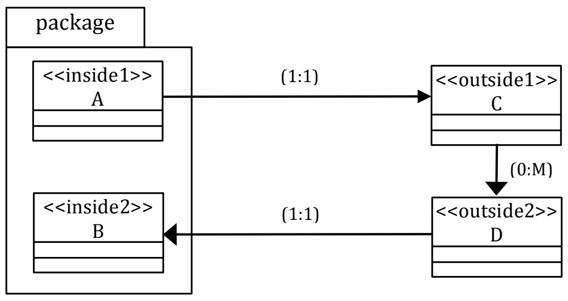


Figure 2.3: Circular Dependency between Packages

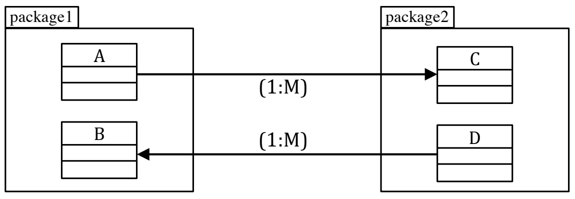


Figure 2.4: Weak Circular Dependency between Packages

Listing 2.3 shows the definition of our SCD antipattern. This antipattern describes a path that creates a dependency cycle. The path starts from a vertex in a package (in1),

```
1  // circular dependencies between packages
2  motif cd
3  select inside1,inside2,outside1,outside2
4  where "inside1.namespace==inside2.namespace"
5     and "inside1.namespace!=outside1.namespace"
6     and "inside1.namespace!=outside2.namespace"
7  connected by outgoing(inside1>outside1)[1,1]
8     and incoming(outside2>inside2)[1,1]
9     and path(outside1>outside2)[0,*]
10 group by "inside1.namespace"
```

Listing 2.3: Circular Dependencies between Packages (CD)

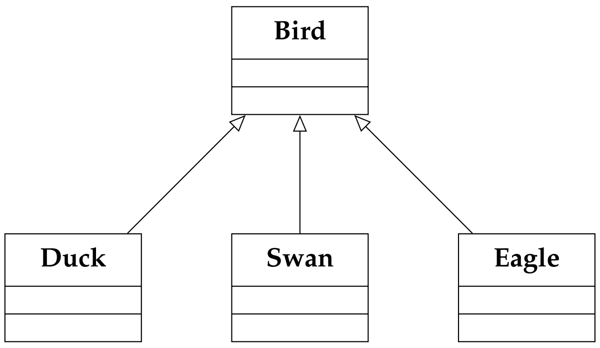


Figure 2.5: Abstraction Example

passes through vertices in other packages (out1, out2), and finally returns back to a vertex in the original package (in2). The vertices (in1, in2) and (out1, out2) may or may not be the same. Package naming constraints are defined on vertices (line 4). The cardinality constraints on outgoing and incoming paths are set to 1 so that those paths contain a single edge. The cardinality constraint [0,*] means a path that may be empty, for instance, where (out1, out2) are the same vertex.

2.3.3 Subtype Knowledge

In object-oriented design, supertypes represent relatively abstract concepts while subtypes represent relatively concrete objects. For example, the UML class diagram as shown in figure 2.5 has a supertype `Bird`, which has common characteristics of birds. The subtypes `Duck, Eagle, Swan` inherit common features of birds and also contain their specific characteristics. With regards to inheritance, Martin (1994) has defined the Dependency Inversion Principle (DIP), which states, "abstractions should not depend upon details. Details should depend upon abstractions".

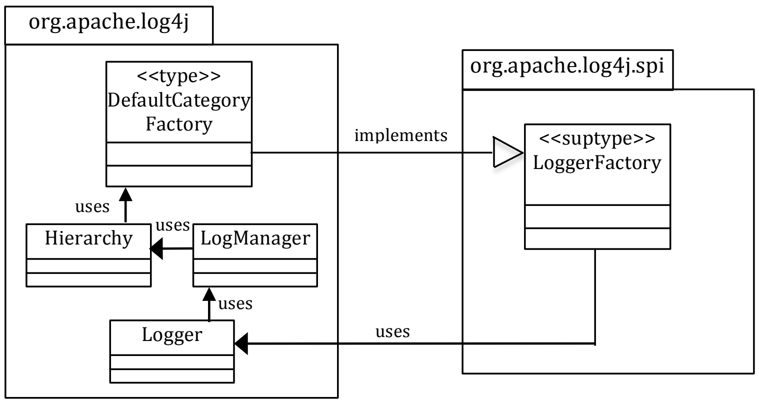


Figure 2.6: Example of Subtype Knowledge Antipattern[4]

In the subtype knowledge (STK) antipattern (Riel, 1996) as shown in figure 2.7 a supertype either directly or indirectly uses its subtype. This implies that we cannot use super and subtypes in isolation. In the presence of STK, separating supertypes and subtypes in different namespaces results in a circular dependency between namespaces. In addition, instability in subclasses (less abstract) causes instability in super classes that tend to be more abstract. Instances of this antipattern violate the dependency inversion principle.

Example

Figure 2.6 shows an instance of the STK antipattern in `Log4J-1.2.15`, where a supertype `org.apache.log4j.spi.LoggerFactory` indirectly uses its subtype `org.apache.-log4j.DefaultCategoryFactory`. Listing 2.4 shows the source code that creates this antipattern instance. This antipattern instance also creates a circular dependency between `org.apache.log4j` and `org.apache.log4j.spi` packages.

Definition

Figure 2.7 shows the visual definition of the STK antipattern. In this definition, a subtype inherits from a supertype either directly or indirectly. In a similar way, a supertype has a *uses* relationship with its subtype either directly or indirectly. Both

[4]Massey Architecture Explorer: `http://goo.gl/1cUHU`

```java
//A subtype inherits from its supertype
package org.apache.log4j;
import org.apache.log4j.spi.LoggerRepository;
...
class DefaultCategoryFactory implements LoggerFactory {
  ...
}
//A supertype uses another class (Logger),
//which uses the subtype
package org.apache.log4j.spi;
import org.apache.log4j.Logger;
...
public interface LoggerRepository {
 public Logger makeNewLoggerInstance(String name);
}
//Logger class uses LogManager
package org.apache.log4j;
...
public class Logger extends Category {
 ...
 static public Logger getLogger(Class clazz) {
    return LogManager.getLogger(clazz.getName());
 }
 ...
}
//LogManager class uses Hierarchy class
package org.apache.log4j;
...
public class LogManager {
 static {
  ...
  Hierarchy h=new Hierarchy(new RootLogger((Level)Level.DEBUG));
  ...
 } ...
}
//Hierarchy class uses DefaultCategoryFactory class
package org.apache.log4j;
...
public class Hierarchy ... {
 private LoggerFactory defaultFactory;
 ...
 public Hierarchy(Logger root) {
    ...
    defaultFactory = new DefaultCategoryFactory();
  } ...
}
```

Listing 2.4: Source code of 5 types creating STK instance in Log4j-1.2.15

sub and supertype can be valid Java reference types. If sub and supertypes are in different packages, we get two antipattern instances, i.e., SCD and STK.

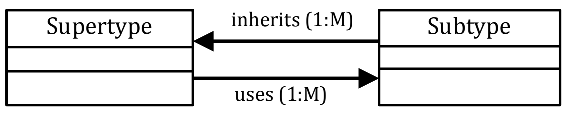


Figure 2.7: Subtype Knowledge

Listing 2.5 shows the definition of the STK antipattern. This antipattern describes two vertex roles *type* and *supertype* after *select* keyword (line 3). Path roles are defined with *connected by* keyword (line 4,5). Path roles define source and target vertex role via ">". The cardinality constraints [1,*] on each path role restricts the minimum path length to 1 and maximum path length to unbound. The default cardinality constraint is [1,*].

The path roles have several constraints (line 6, 7). These constraints are satisfied when the path role instantiates a path with the respective type label. For example, *inherits.type* == '*extends*' evaluates to true only when all edges in the path role *inherits* have a label *extends*. Finally, results are grouped by the *supertype* vertex role (line 8).

```
1 // subtype knowledge
2 motif stk
3 select type,supertype
4 connected by inherits(type>supertype) [1,*]
5    and uses(supertype>type) [1.*]
6 where "inherits.type==‘extends’ || inherits.type==‘implements’"
7    and "uses.type==‘uses’"
8 group by "supertype"
```

Listing 2.5: Subtype Knowledge (STK)

2.3.4 Abstraction Without Decoupling

The Dependency Inversion Principle (DIP) states that "high level modules should not depend upon low level modules. Both should depend upon abstractions" (Martin, 1994). This means a class should depend on an abstract type rather than on a concrete class. However, these abstract types still have to be instantiated using concrete classes. In the abstraction without decoupling (AWD) antipattern (Figure 2.9), a client depends on a service (supertype) and an implementation of the service (subtype) at the same time, thus violating the dependency inversion principle. This situation could be avoided, had the client only depended on the service and not on the implementation.

This antipattern has several problems, such as that the client code needs to be changed whenever the service implementation changes. Test cases associated with that specific service implementation also need to be changed. Finally, this antipattern hampers the ability to dynamically swap different service implementations, which is an important feature in modern modular platforms.

Example

Figure 2.8 shows an instance of the AWD antipattern in `Log4J-1.2.15`, where a client class `org.apache.log4j.jdbc.JDBCAppender` uses an abstract class `org.apache.-log4j.Layout` and its implementation class `org.apache.log4j.PatternLayout` at the same time. Listing 2.6 shows the source code that creates this antipattern instance. This hard-coded dependency on `PatternLayout` may lead to a runtime exception, if we provide `JDBCAppender` a different implementation of `Layout`. This particular dependency also causes other design problems related to orthogonality (Dietrich, 2013).

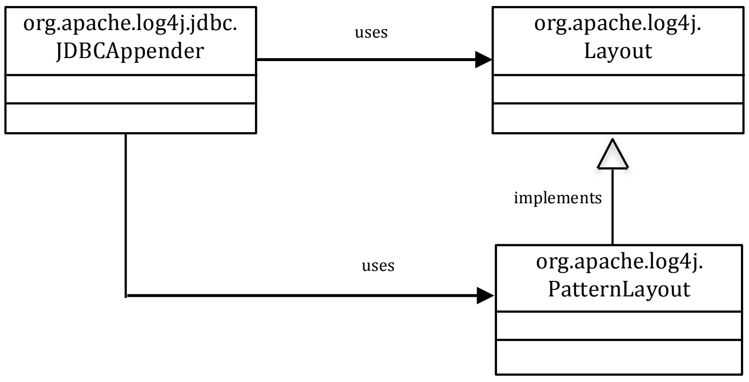


Figure 2.8: Example of Abstraction Without Decoupling Antipattern[5]

Definition

Figure 2.9 shows the visual definition of the AWD antipattern. In this figure, there is a single edge that connects *client* and *supertype*, while there could be an indirect path between the client and the subtype. Listing 2.7 shows the definition of the antipattern as a graph query. In this definition, constraints are defined on vertex and path roles (line 3,4 and 8,9 respectively). Path length constraints are omitted for some path roles. In that case the default path length constrains [1,*] are used.

[5]Massey Architecture Explorer: `http://goo.gl/JxAoW`

```java
package org.apache.log4j.jdbc;
...
public class JDBCAppender
 extends org.apache.log4j.AppenderSkeleton ... {
 ...

 public void setSql(String s) {
    sqlStatement = s;
    if (getLayout() == null) {
        this.setLayout(new PatternLayout(s));
    }
    else {
        ((PatternLayout)getLayout()).setConversionPattern(s);
    }
  }
}

package org.apache.log4j;
...
public abstract class AppenderSkeleton ... {
 ...
 public void setLayout(Layout layout) {
  this.layout = layout;
 }
 ...
}
```

Listing 2.6: Source Code Creating an AWD Instance in Log4J-1.2.15

2.3.5 Degenerated Inheritance

In software development, duplication is considered to be dangerous at all levels whether it is in requirements, architecture, source code or documentation. Whenever an artefact is changed, all copies of the artefact must be detected and changed. This makes the maintenance of a system very expensive. To avoid duplication, it is recommended to follow a principle called DRY (Don't Repeat Yourself)[6], which states, "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". The degenerated inheritance (DEGINH) antipattern (Sakkinen, 1989) as shown in figure 2.11 appears in a program when there are multiple inheritance paths from a subtype to its supertype. In Java programs, this can be introduced by the use of multiple inheritance through interfaces. This antipattern creates duplication in the program structure and makes it difficult to separate subtypes from their supertypes.

[6]http://c2.com/cgi/wiki?DontRepeatYourself

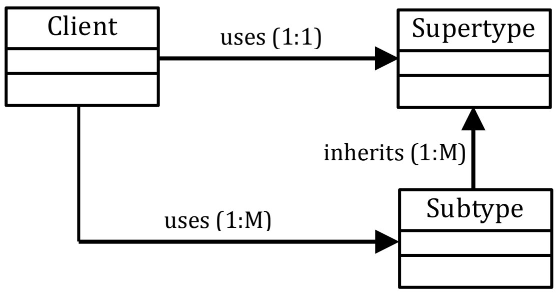


Figure 2.9: Abstraction Without Decoupling

```
motif awd
select client, service, service_impl
where "!client.abstract" and "service.abstract"
   and "!service_impl.abstract"
connected by inherits(service_impl>service)
   and service_invocation(client>service)[1,1]
   and implementation_dependency(client>service_impl)
where "inherits.type=='extends' || inherits.type=='implements'"
   and "service_invocation.type=='uses'"
   and "implementation_dependency.type=='uses'"
group by "client" and "service"
```

Listing 2.7: Abstraction Without Decoupling (AWD)

Example

Figure 2.10 shows an instance of the DEGINH antipattern in Log4J-1.2.15, where a type `org.apache.log4j.jdbc.JDBCAppender` has two paths to its supertype `org.apache.-log4j.Appender`. Listing 2.8 shows the source code that creates this antipattern instance.

Definition

Figure 2.11 shows the visual definition of the DEGINH antipattern. In this definition, two paths originate from a subtype to a supertype. The length of both paths is lower bound 1 and no upper bound (1:M). The definition of the graph query is shown in

[7]Massey Architecture Explorer: `http://goo.gl/bCX0g`

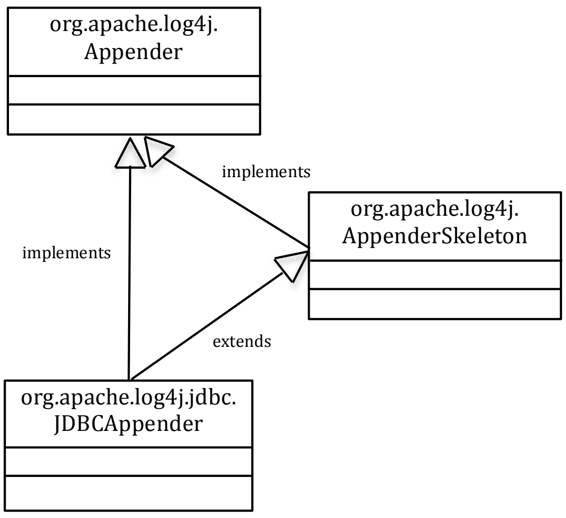


Figure 2.10: Example of Degenerated Inheritance Antipattern[7]

```java
package org.apache.log4j.jdbc;
...
public class JDBCAppender
    extends org.apache.log4j.AppenderSkeleton
    implements org.apache.log4j.Appender {
   ...
}
```

Listing 2.8: Source Code of JDBCAppender Creating DEGINH Instance in Log4j-1.2.15

listing 2.9. In this query, there is a constraint on a vertex role, which states that the supertype cannot be `java.lang.Object` (line 4), because in Java all classes inherit from the `Object` class.

The *find all* flag (line 6) indicates that the Guery engine should find all mappings of the respective path roles. By default, the engine finds all possible vertex bindings, that is, vertex roles are mapped to vertices in the graph but for each vertex mapping only a single path is initialised (path role to sequence of edges mapping). This restriction is sometimes necessary for scalability reasons.

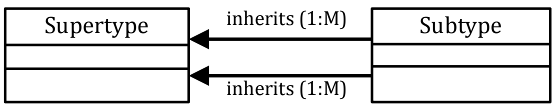


Figure 2.11: Degenerated Inheritance

```
//degenerated inheritance
motif deginh
select type,supertype
where "supertype.fullname!='java.lang.Object'"
connected by inherits1(type>supertype)
  and inherits2(type>supertype) find all
where "inherits1.type=='extends'||inherits1.type=='implements'"
  and "inherits2.type=='extends'||inherits2.type=='implements'"
where "inherits1!=inherits2"
group by "supertype"
```

Listing 2.9: Degenerated Inheritance (DEGINH)

2.4 Detecting Opportunities - Scoring Edges

Scoring is a technique used to determine the relative importance of vertices or edges in a graph. We have investigated several algorithms that use a scoring concept. These include Google's PageRank (Brin and Page, 1998), Betweenness Centrality (Freeman, 1977), Edge Betweenness (Girvan and Newman, 2002). A common pattern in scoring algorithms is that they walk through the network and count how often a vertex or edge is visited. Table 2.2 shows a summary of different scoring mechanisms.

In our approach, an antipattern instance corresponds to a subgraph in the dependency graph. Due to the presence of large numbers of antipattern instances in the dependency graph, it may become very difficult to remove all antipattern instances individually from the graph. Therefore, we have used a scoring algorithm, which assigns a high weight to edges that participate in large numbers of antipattern instances. The algorithm associates a number with each edge indicating in how many antipattern instances this edge occurs. This means that the removal of a high-scored edge would likely remove multiple antipattern instances. Thus, this scoring mechanism allows us to identify high-impact refactoring opportunities, i.e., by performing a few refactorings, a large number of antipattern instances can be removed (Dietrich et al., 2012). In our scoring algorithm, edges are scored and traversal is performed on paths in antipattern instances.

Score/Metric	Scores what	Paths visited
Google PageRank	vertices	random paths
Betweenness centrality	vertices	shortest paths
Edge Betweenness	edges	shortest paths
Antipattern Score	edges	paths in antipatterns

Table 2.2: Comparison of Different Scoring Mechanisms

Figure 2.12 shows an example of the dependency graph of a program. There are five classes in the program namely A, B, C, D, E in four packages named as package1, package2, package3 and package4. This program has two types of antipattern instances i.e., SCD and STK. The following paths represent antipattern instances:

- An SCD antipattern instance is represented by the path $A \rightarrow_{extends} B \rightarrow_{uses} D \rightarrow_{uses} A$.
- An SCD instance caused by $B \rightarrow_{uses} D \rightarrow_{uses} E$.
- An STK antipattern instance is represented by the path $B \rightarrow_{uses} D \rightarrow_{uses} A \rightarrow_{extends} B$.

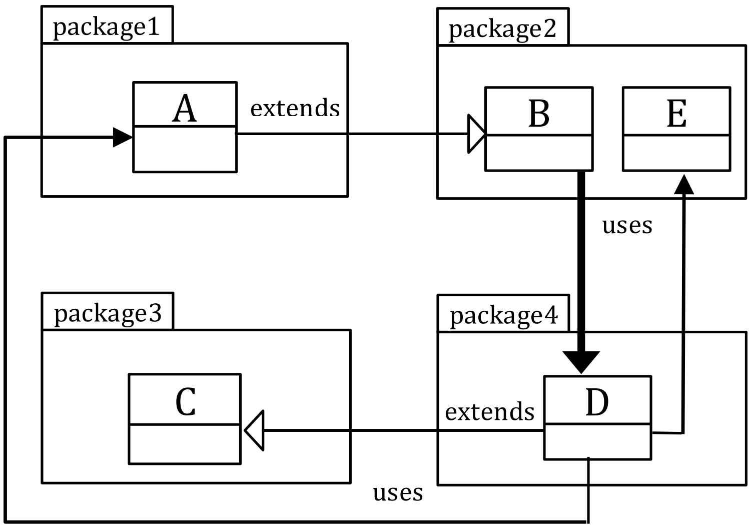


Figure 2.12: Example Program's Dependency Graph

In this example, the edge $B \rightarrow_{uses} D$ gets a score of three. The rest of the edges either get one or two. All of these antipattern instances can be removed either by redirecting the edge $B \rightarrow_{uses} D$ to $B \rightarrow_{uses} C$ or by removing the edge $B \rightarrow_{uses} D$.

In our algorithm, we have computed those edges with the highest score. If multiple edges get the same score, we sort them by the fully qualified name of the start and the end vertex to make the process reproducible.

2.5 Dependency Classification

In this section, we define the meaning of a dependency in a Java program. This will later be used in order to classify dependencies that our refactoring algorithm will remove. A dependency has different types. Our classification of a dependency is derived from earlier works on dependencies (Lakos, 1996; Melton and Tempero, 2007a). Listing 2.10 shows different types of dependencies in a Java program. In this listing class A depends on other types B, C, D, E, F, G, H and System. In this context, we call the dependent class A the **source type**, and the rest of dependency classes **target types**.

```java
public class A extends B implements C {
   private D object = new E();
   public F m(G obj) throws H {
      System.out.println(obj.toString());
      ...
      return obj.getF();
   }
}
```

Listing 2.10: Java Source Code Creating Dependencies

We can broadly classify a dependency relationship between a source and a target type into the following nine categories:

1. Variable Declaration (VD): The target type is used to declare a variable or a field, for example, $A \rightarrow_{uses} D$.
2. Method Return Type (MRT): The target type is used as a return type of a method in the source type, for example, $A \rightarrow_{uses} F$.
3. Method Parameter Type (MPT): In this case the target type is used as a method parameter in the source type, for example, $A \rightarrow_{uses} G$.
4. Method Exception Type (MET): The target type is used as an exception type using *throws* keyword, for example, $A \rightarrow_{uses} H$.
5. Constructor Invocation (CI): A target type constructor is invoked through *new* keyword, for example, $A \rightarrow_{uses} E$.

6. Static Member Invocation (SMI): When a static member of a class (field or method) is invoked on the target type, for example, $A \rightarrow_{uses} System$.

7. Superclass (SC): The target type is used as a supertype through the *extends* keyword, for example, $A \rightarrow_{extends} B$.

8. Interface (IN): The target type is used as an interface through *implements* keyword, for example, $A \rightarrow_{implements} C$.

9. Other: In this category, references to a class, such as *B.class* and *object instanceof E* are included.

In the dependency graph, an edge can represent three types of relationships: *uses*, *extends* and *implements*. Categories from 1-6 and 9 are represented as a *uses* edge. SC relationship is represented by an *extends* edge, while an *implements* edge represents IN relationships.

2.6 The Dataset

In our experiments, we have analysed a large dataset of programs. These programs are collected from the Qualitas Corpus (Tempero et al., 2010) - a large collection of open source programs. The main reason for choosing this particular dataset is that it contains a large variety of real-world programs that gives us the opportunity to apply our techniques on a broad spectrum of programs. Programs in the dataset include real-world applications, end user libraries and tools. Furthermore, the dataset is readily available and all programs in the corpus are written in Java language. There exist other datasets for empirical studies, such as Software-artifact Infrastructure Repository (Do et al., 2005), Sourcerer (Bajracharya et al., 2006), and Purdue Benchmark Suite (Grothoff et al., 2007). Among these datasets the Qualitas Corpus turns out to be most comprehensive and most widely used dataset.

We have used the Qualitas Corpus version 20101126. This version contains 106 programs. The corpus comes in "r" (recent) and "e" (evolution) editions. In the "e" edition, many programs have multiple versions of the same program. This corpus is useful for studies related to evolution of software systems. Our aim is not to study software evolution, therefore we have used the "r" edition, which contains only the most recent version of every program.

We have excluded 16 programs from the dataset based on two criteria: size and configurability in Eclipse. The JRE-1.6 was skipped due to the large number of classes and relationships (17253, 173911 respectively). Although Guery is scalable enough to find antipatterns in JRE, we had to repeat this analysis multiple times to run simulations. The other 15 skipped programs were plugin based, i.e., each program had multiple plugins/modules that were difficult to normalise into a single Eclipse project[8]. Also, these programs make use of custom class loading and the definition of dependencies through configuration files, which is not reflected in the models we use. However, we believe that the resulting dataset of 92 programs still has a wide variety of real-world applications sufficient for the evaluation purpose.

The 92 programs from the dataset have been manually configured as Eclipse projects and are loaded into a workspace. In order to normalise all projects, we created a folder structure in each Eclipse project. This includes a `src` folder, which contains source code of a program. The `bin` folder contains the compiled output files. The `lib` folder includes all the required external library files. The `test` folder contains the source code of tests, if included with the original source code of the program. We configured all projects to use JRE 1.6. This created several problems for those programs that were configured with JRE 1.4 or before. In particular, some of the existing programs used `enum` as an identifier, whereas in JRE 1.6, `enum` is used as a keyword. For those projects, we manually renamed all such identifiers, with the help of search and replace, to remove compilation errors.

[8]The skipped programs are: eclipse- 3.6, netbeans-6.9.1, exoportal-v1.0.2, gt2-2.7-M3, ivatagroupware-0.11.3, jboss-5.1.0, jtopen-7.1, maven-3.0, myfaces-core-2.0.2, roller-4.0.1, spring-framework-3.0.5, struts-2.2.1, tapestry-5.1.0.5

ant-1.8.1	antlr-3.2	aoi-2.8.1
argouml-0.30.2	aspectj-1.6.9	axion-1.0-M2
azureus-4.5.0.4	c_jdbc_2.0.2	castor-1.3.1
cayenne-3.0.1	checkstyle-5.1	cobertura-1.9.4.1
colt-1.2.0	columba-1.0	compiere-330
derby-10.6.1.0	displaytag-1.2	drawswf-1.2.9
drjava-stable-20100913-r5387	emma-2.0.5312	findbugs-1.3.9
fitjava-1.1	fitlibraryforfitnesse-20100806	freecol-0.9.4
freecs-1.3.20100406	galleon-2.3.0	ganttproject-2.0.9
heritrix-1.14.4	hibernate-3.6.0-beta4	hsqldb-2.0.0
htmlunit-2.8	informa-0.7.0-alpha2	ireport-3.7.5
itext-5.0.3	jag-6.1	james-2.2.0
jasml-0.10	jasperreports-3.7.3	javacc-5.0
jchempaint-3.0.1	jedit-4.3.2	jena-2.6.3
jext-5.0	jFin_DateMath-R1.0.1	jfreechart-1.0.13
jgraph-5.13.0.0	jgraphpad-5.10.0.2	jgrapht-0.8.1
jgroups-2.10.0	jhotdraw-7.5.1	jmeter-2.4
jmoney-0.4.4	joggplayer-1.1.4s	jparse-0.96
jpf-1.0.2	jrat-0.6	jrefactory-2.9.19
jruby-1.5.2	jspwiki-2.8.4	jsXe-04_beta
jung-2.0.1	junit-4.8.2	log4j-1.2.16
lucene-2.9.3	marauroa-3.8.1	megamek-0.35.18
mvnforum-1.2.2-ga	nakedobjects-4.0.0	nekohtml-1.9.14
openjms-0.7.7-beta-1	oscache-2.4.1	picocontainer-2.10.2
pmd-4.2.5	poi-3.6	pooka-3.0-080505
proguard-4.5.1	quartz-1.8.3	quickserver-1.4.7
quilt-0.6-a-5	rssowl-2.0.5	sablecc-3.2
sandmark-3.4	squirrel_sql-3.1.2	sunflow-0.07.2
tomcat-7.0.2	trove-2.1.0	velocity-1.6.4
webmail-0.7.10	weka-3.7.2	xalan-2.7.1
xerces-2.10.0	xmojo-5.0.0	

Table 2.3: The Dataset

Chapter 3

Dependency-Breaking Refactorings

In the previous chapter, we have described several antipatterns that compromise modularity. We also discussed several kinds of dependencies between software artefacts. We then discussed a scoring mechanism which identifies critical dependencies between classes that point to high-impact refactoring opportunities. In this chapter, we embark upon investigating refactoring techniques that are widely used to decouple classes. Each refactoring is described with an example and a discussion of its pros and cons. Finally, we choose a set of refactorings to be used in experiments based on several requirements. In the next chapters, we describe the process for applying these refactorings to break unwanted dependencies between classes from programs.

3.1 Overview

Dependencies between classes represent relationships that occur because one class uses services of another class. For example, a class A depends on a class B when A inherits from B, has an attribute whose type is B, or has a method that calls methods declared in B. In this case, A is called *source* while B is called *target*. This dependency means that A cannot function without B, and A cannot be reused without B. In Java, if the target is missing, a compile time error is generated. At runtime, the missing target type results in the generation of `ClassNotFoundException`[1] or `NoClassDefFoundError`[2].

In the previous chapter, we divided dependencies between classes into nine categories. Next, we look into several levels of a dependency. For instance, when a method of the class A calls a method of the class B, this automatically creates a dependency between

[1]There are certain "weak" dependencies that cannot be verified by the compiler, in particular introduced through reflection.

[2]An error is generated when JVM or class loader cannot find the definition of a class.

A and B. If A and B happen to be in different packages, this will create a dependency between packages. Similarly, when two packages are deployed in different jar files to improve maintainability, this will create a dependency between these two jars. In this thesis, we restrict our focus to dependencies on the class and the package level because the antipatterns we defined in the previous chapter focus on these two artefacts of programs.

At times, it becomes necessary to decouple dependencies in programs to improve several software quality characteristics such as modularity, reusability, and testability. For example, external dependencies (such as file systems, threads, memory) must be broken in order to test the program in a controlled environment. Several techniques have been used in the literature to break dependencies for achieving the aforementioned quality attributes in programs. These techniques are basically refactorings that can be used to break dependencies in programs. Notable works in this area include Martin (1994); Fowler (2004); Feathers (2004); Stal (2008), and Osherove (2009). In the next sections, we explain dependency-breaking refactorings on the class and the package level. We follow a simple template where we describe a refactoring with an example and in some cases we explain the example through a visual diagram before and after refactoring. The list of refactorings is as follows:

Package Level Refactorings

- Move Class
- Split Package
- Merge Package

Class Level Refactorings

- Adapt Parameter
- Extract Interface
- Dependency Injection
- Service Locator
- Type Generalisation
- Static Members Inlining

3.2 Package Level Refactorings

3.2.1 Move Class

The move class refactoring is commonly used to break dependency cycles between packages (RefactoringCatalog, 1999). This refactoring is rather simple and robust tool support exists to automate it. Modern IDEs such as Eclipse, Netbeans, and IntelliJ IDEA contain a built-in support for the move refactoring. These tools automatically fix references in other classes affected by the move class refactoring.

Another advantage of move refactoring is that it can be completely simulated on the model level. This provides the ability to evaluate the impact of refactoring without changing the source code of programs. For example, on the dependency graph level, where classes are represented as vertices and edges represent relationship between classes, a move class refactoring can be performed by changing labels (namespace) on respective vertices.

Example

Figure 3.1 shows an example of move class refactoring on the model level. In this scenario, we assume that the dependency between class A and B represents a bad dependency and should be removed. A trivial solution of moving the class B from package2 to package1 would resolve the problem. However, in some cases there could be side effects of the move class refactoring.

Consider figure 3.2 as an example where a move refactoring may have negative consequences. In this example, when the class B is moved to package1, we created another dependency from package2 to package1. This dependency occurred because classes C and D depend on the class B. This situation has increased coupling between the packages and decreased cohesion inside package2. Another complication associated with this example could be related to the visibility of elements of the class B. For instance, suppose the class B has several methods that have package level visibility. Moving the class B to package1 will result in compilation error, if there are any references to those methods by class C or D in package2. In this case, we not only move the class B to package1 but also change the visibility of the members affected by the move. We refer to this as an *additional refactoring*. Renaming a class before moving could be another additional refactoring. This refactoring is required if the package, where the class is being moved, contains a class with the same name.

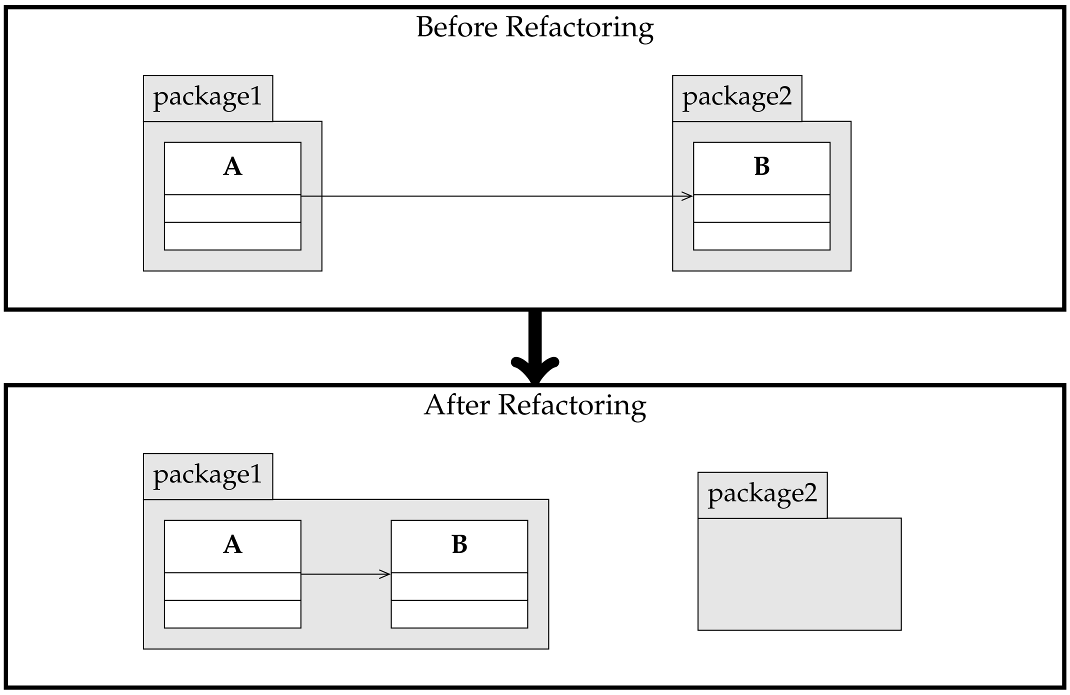


Figure 3.1: Move Class Example

3.2.2 Split Packages

Packages should have high cohesion and low coupling. However, if there is loose cohesion within a package, this reflects that the functionality needs to be split in multiple packages. In order to resolve the problem, areas with loose cohesion within a package should be identified and they become candidates for split refactoring. In a similar way, if a package has too many responsibilities, it should be split. Dietrich et al. (2008) suggested the use of clustering to detect opportunities for splitting packages. The split package refactoring can also be used to break dependency cycles between packages, although dependency cycles that are created through a single path cannot be broken with this refactoring (see Section 2.3.2). This refactoring is also known as split subsystems (Stal, 2008).

Example

Figure 3.3 shows an example of split packages. In this example, move class refactoring is used to move the cohesive classes. This refactoring is useful when there is weak circular dependency between packages, i.e., there are two individual paths that create the dependency between packages.

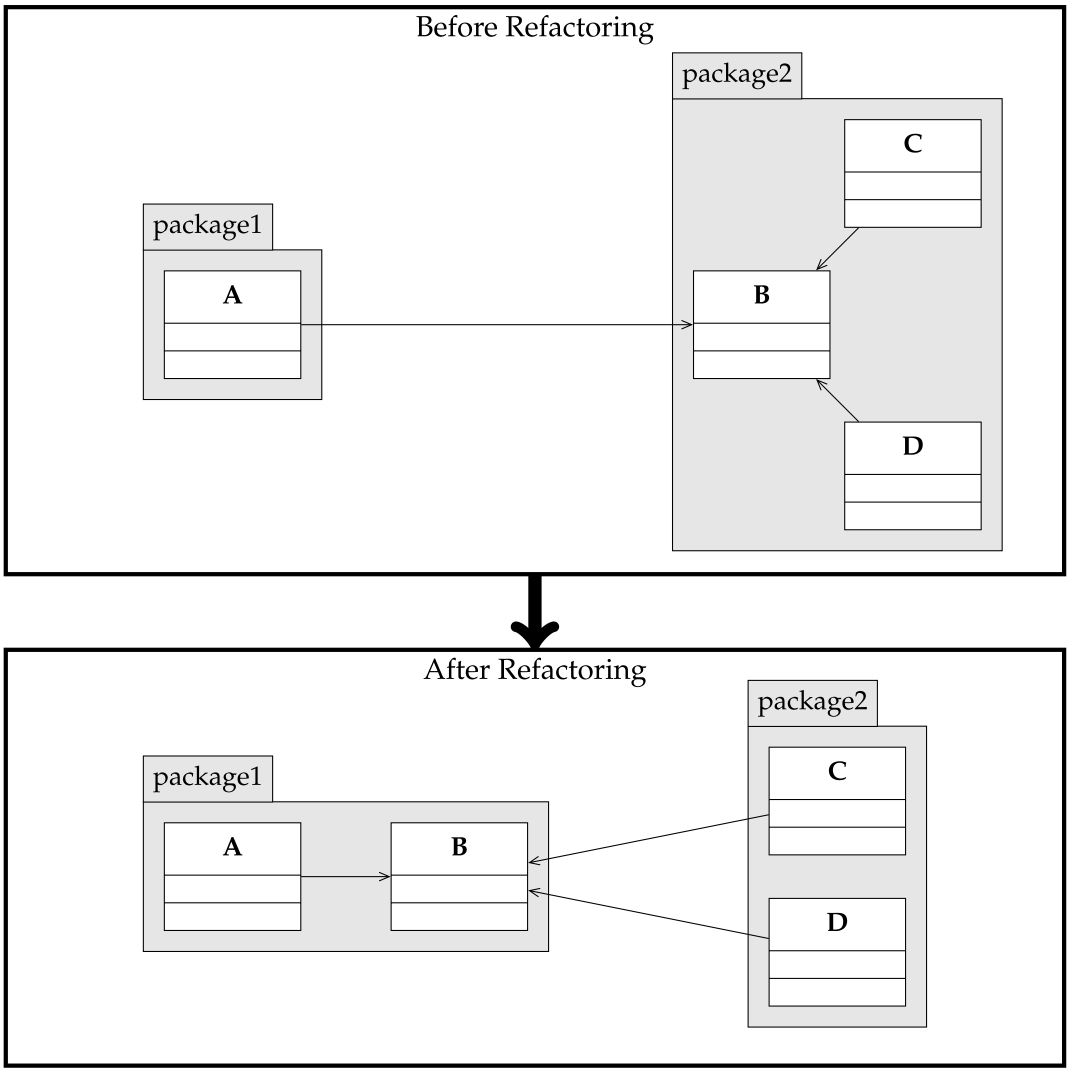


Figure 3.2: Move Class Example

3.2.3 Merge Packages

When there is high coupling between two packages, it is desirable to merge them (Stal, 2008). In this technique, classes inside two packages are put together in a single package. This refactoring eliminates the dependency cycles between the packages. The resulting merged package might then need to be divided based on other criteria, such as cohesion metrics. In figure 3.3, if all classes use each other, we could move classes A and B to package2 and delete package1 to remove the dependency cycle.

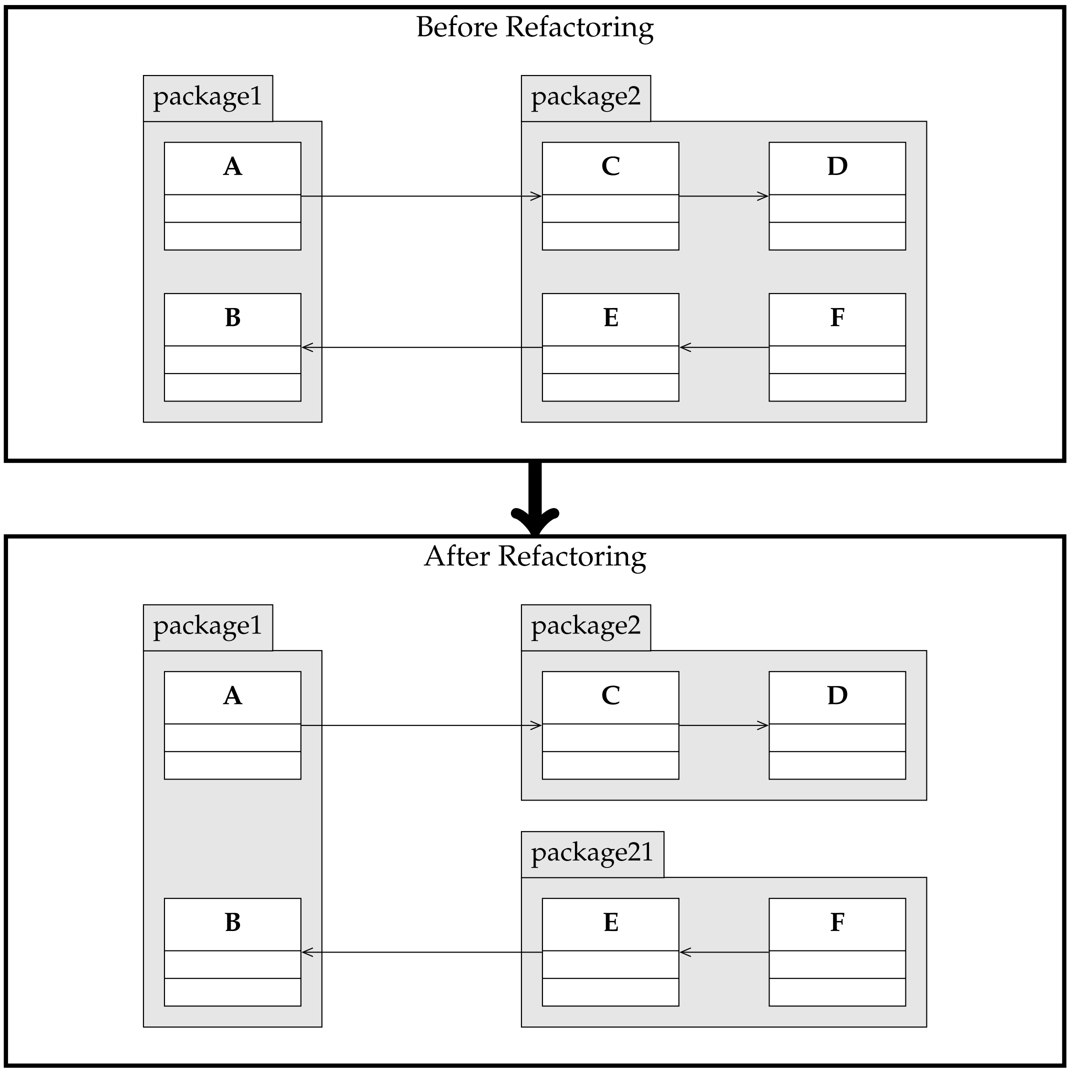


Figure 3.3: Split Packages Example

3.3 Class Level Refactorings

3.3.1 Adapt Parameter

This refactoring applies to method parameters. In this refactoring, a dependency to the target type[3] is broken by removing the target type from the method parameter and replacing it with a new interface, whose implementation provides the required functionality (Feathers, 2004). This new interface does not have to be implemented by the target type, instead an adapter is used. This is basically an indirection refactoring, which can be used when it is difficult to extract an interface from the existing parameter

[3]A type refers to a non-array Java reference type.

type. This is an invasive refactoring[4] as we need to introduce a new interface and also need to change the parameter in the method signature and this could break the client code.

Example

Listing 3.1 shows an example of the adapt parameter refactoring. In this example, there is a dependency from `MyServletRequester` (source) to `HttpServletRequest` (target). If we want to test the `populate` method, we need to provide an implementation of `HttpServletRequest`, which has more than twenty methods. In this situation, it is difficult to create a mock object for testing purposes. In order to avoid this situation, a small interface `ParameterSource` is created to be used in the method parameter. By using this interface, two different implementations can be created, i.e., each for production and testing.

Figure 3.4 shows the class diagram before and after the refactoring. The solid arrow represents the dependency relationship[5]. The dependency relationship is labelled as a *uses* relationship. The triangular arrow head represents an inheritance relationship and is labelled as *extends* or *implements*. There are two boxes in the diagram, which represent before and after refactoring. In the diagram (After Refactoring), the *adapter pattern* (Gamma et al., 1994) is used to break the dependency to `HttpServletRequest`. The class `ServletParameterSource` serves as *adapter* or *wrapper* and the class `HttpServletRequest` serves as *adaptee*.

```
//Before adapt parameter refactoring
public class MyServletRequester { ...
   public void populate(HttpServletRequest request) {
      String[] values = request.getParameterValues(pageStateName);
      ...
   } ...
}

//After adapt parameter refactoring
public class MyServletRequester { ...
   public void populate(ParameterSource source) {
      String[] values = source.getParameterValues(pageStateName);
      ...
```

[4]We quantify the invasiveness property by the number of primitive refactorings required to perform a composite refactoring.

[5]In the dependency relationship, the target class is used to declare fields, local variables, method return types, method exception types and so on.

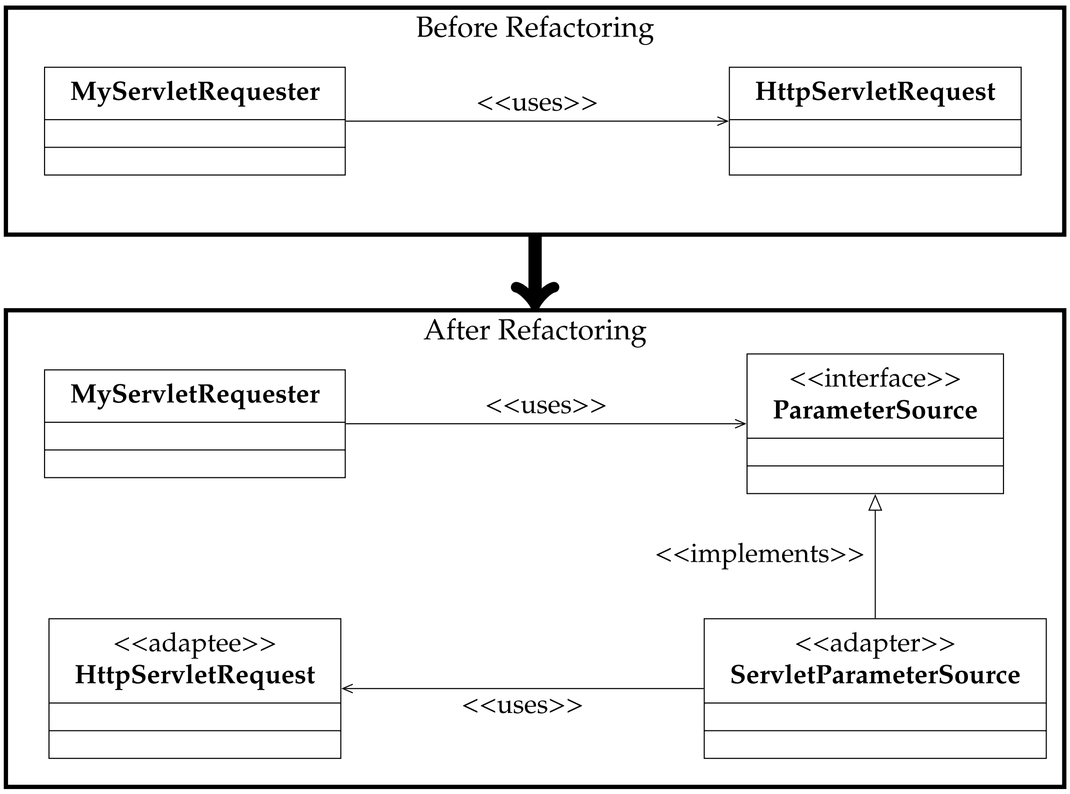


Figure 3.4: The Adapt Parameter Refactoring

```
   } ...
}

public interface ParameterSource {
   public String[] getParameterValues(String name);
}

public class ServletParameterSource implements ParameterSource
{
   private HttpServletRequest request;
   public ServletParameterSource(HttpServletRequest request) {
      this.request = request;
   }
   public String[] getParameterValue(String name) {
      return request.getParameterValues(name);
   }
}
```

Listing 3.1: The Adapt Parameter Refactoring

3.3.2 Extract Interface

This is a commonly used refactoring to break dependency from the source to the target class. In order to perform this refactoring, we have to identify the minimum possible interface (public methods) of the target class used in the source class. After that, a new interface can be created from the public methods identified and this interface can be used as the replacement for the target class. The target class should now implement the new interface. The extract interface refactoring requires human intervention to assign a name to the new interface and to create alternative implementations for testing purposes (Fowler, 1999).

Example

Listing 3.2 shows an example where the extract interface refactoring is used to break the dependency from the target class (`Employee`). In this example, the `TaxCalculator` class is used to calculate taxes of employees. This class needs the base tax rate for an employee to calculate the tax. The `calculateTax` method uses only a single method (`getBaseTaxRate`) from the `Employee` class. The `Employee` class has many other features than `getBaseTaxRate`. Therefore, this method can be used to define a new interface as shown in figure 3.5. Furthermore, the `Employee` class now implements the `Taxable` interface. Similarly, an alternative implementation of `Taxable` can be created as a mock implementation for testing the `calculateTax` method.

```java
//Before extract interface refactoring
public class TaxCalculator { ...
   public double calculateTax(Employee emp, int salary) {
      int baseRate = emp.getBaseTaxRate();
      return baseRate * salary;
   } ...
}
//After extract interface refactoring
public class TaxCalculator { ...
   public double calculateTax(Taxable emp, int salary) {
      int baseRate = emp.getBaseTaxRate();
      return baseRate * salary;
   } ...
}
public interface Taxable {
   public double getBaseTaxRate();
}
```

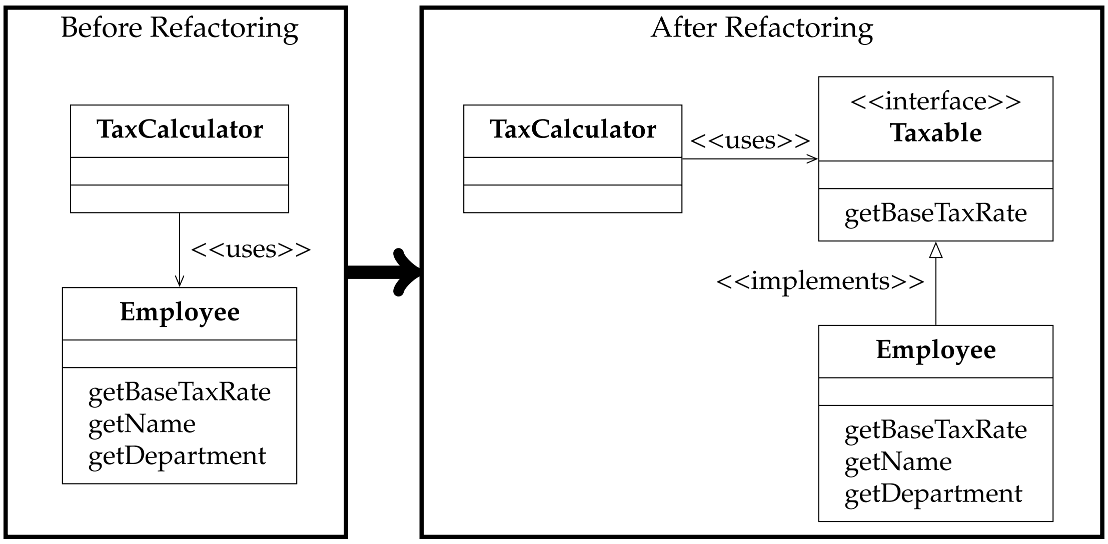


Figure 3.5: Extract Interface Refactoring

```java
public class Employee implements Taxable {
    ...
}
```

Listing 3.2: Extract Interface Refactoring

3.3.3 Dependency Injection

Dependency injection or inversion of control is a powerful technique for decoupling the dependency between classes. This technique is based on the dependency inversion principle postulated by Martin (1994). The term dependency injection was coined by Fowler (2004). In this technique, the client class does not know about a specific implementation class at compile time. The implementation class is injected at runtime by an assembler object as shown in figure 3.6 (After Refactoring). The assembler object can use any of the existing dependency injection frameworks to inject dependencies at runtime. These frameworks include Spring (Johnson et al., 2005), OSGi Declarative Services (Walls, 2009), Google Guice (Vanbrabant, 2008), and PicoContainer (Hammant and Hellesy, 2003). These frameworks can be used to define the configuration of implementation types required by an application. A benefit of these frameworks is that we can use alternative implementations of a service at runtime without restarting or recompiling the application.

Example

In listing 3.3, the `BankingApp` class (client), depends on an interface `ObjectStore` (representing a service) and its implementation type `XMLObjectStore` (service implementation) at the same time (figure 3.6 (Before Refactoring)). This has several problems. For example, when the service implementation is changed then the client code needs to be changed. Test cases associated with that specific service implementation also need to be changed. This situation could be avoided, had the client depended only on the service and not on the implementation. In this case, whatever service implementation is used, there is no need to change the client code or the test cases. This dependency from the client to the implementation class can be broken by using the dependency injection. According to Fowler (2004), there are three types of dependency injection, i.e., constructor injection, setter injection, and interface injection.

```java
//Before refactoring
public class BankingApp { ...
   private ObjectStore store;
   public void backup( ) {
      store = new XMLObjectStore ( ) ;
      ...
   } ...
}
```

Listing 3.3: Source Code of Coupling Example

Constructor Injection

This refactoring is also known as parametrize constructor (Feathers, 2004). In this technique, the reference to object creation inside a method is replaced with a parameter in the constructor of the client class. This parameter represents the type whose implementation will be used by the client class. In this way, we are able to remove the hard-coded dependency and now can inject any alternative dependency at runtime. The class that will inject the dependency in the client class needs to use any dependency injection framework to specify and configure the implementation types.

```java
//After refactoring
public class BankingApp { ...
   private ObjectStore store;
   public BankingApp (ObjectStore s) {
```

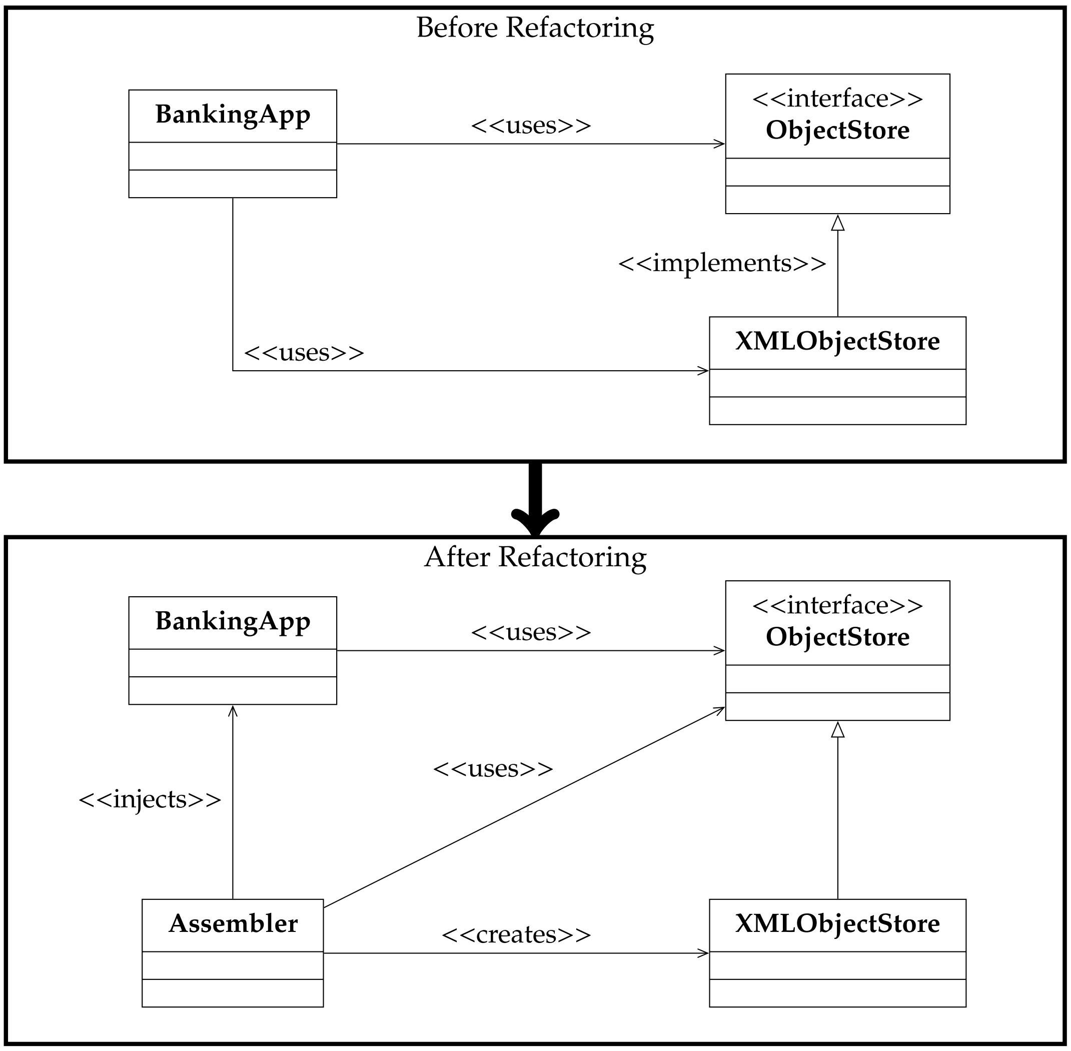


Figure 3.6: Example of Dependency Injection

```
        this.store = s;
    }
    public void backup( ) {
        ...
    } ...
}
```

Listing 3.4: Source Code After Constructor Injection

Setter Injection

This technique to inject dependencies is similar to constructor injection. However, the difference lies in the declaration of setter methods inside the client class to inject dependencies from outside. Listing 3.5 shows the code after introducing the setter method in the client class to receive a dependency.

```java
//After refactoring
public class BankingApp { ...
   private ObjectStore store;
   public void setObjectStore(ObjectStore s) {
      this.store = s;
   }
   public void backup( ) {
      ...
   } ...
}
```

Listing 3.5: Source Code After Setter Injection

Interface Injection

In this technique, a new interface is defined and used to inject dependencies. First, a new injector interface is defined for the purpose of injecting `ObjectStore`. This interface is defined with a generic type parameter so that it can be used to inject any object. After that, this new interface is implemented by any class that requires `ObjectStore`, such as the `BankingApp` class as shown in listing 3.6.

```java
public interface Injector<T> {
   public void inject(T target);
}
public class BankingApp implements Injector<ObjectStore> { ...
   private ObjectStore store;
   public void inject(ObjectStore target) {
      this.store = target;
   }
   public void backup( ) {
      ... } ...
}
```

Listing 3.6: Source Code After Interface Injection

3.3.4 Service Locator

The service locator pattern can also be used to address the decoupling problem. In this pattern, service implementation classes are decoupled from their client classes. Fowler (2004) describes the service locator pattern as a registry that is used to look up instances of implementation classes. This pattern is particularly useful when there exist alternative service implementations and they need to be replaced or upgraded.

For example, Java Database Connectivity (JDBC) API has a Driver Manager, which is used to establish JDBC connections. The JDBC drivers are supplied by different parties. It is not desirable to rely on a specific driver implementation, instead a string encoded URL is used, which provides a connection string that is then accepted by a certain driver implementation. For instance, the abstract `Connection` type can be instantiated as follows, avoiding references to particular implementation: `Connection c = DriverManager.getConnection(aURL)`. In this case, a specific driver implementation is not hard-coded by the client class, rather a dependency is established through the JDBC driver manager. The `DriverManager` class uses name matching based on the URL to locate the driver implementation. When the above method is called, it iterates over all the registered drivers and finds an appropriate driver for the connection string aURL. This searching is performed by a method `getDriver(aURL)`, which further invokes a method `acceptURL(aURL)` to iterate through the registered drivers and return the first driver implementation that could connect to the database. Service Locators are also used in Common Object Request Broker Architecture (CORBA) and Java Naming and Directory Interface (JNDI). Java Service Loader is a generic solution for custom service loaders that is part of the JDK.

According to Fowler, both service locators and dependency injection can be used to break dependencies from the client class, however, there is a subtle difference between the two techniques. In the former, the client class pulls a required service from a registry, while in the latter the required service is pushed (injected) into the client class. The use of service locator to break a dependency is not invasive as compared to the dependency injection, which often requires the generation of additional methods (setters) and constructors in the client class and may conflict with encapsulation.

Example

Consider listing 3.7 as an example of the service locator pattern. In this example, the `XMLObjectStore` object is instantiated through a Java service loader class `java.util.ServiceLoader`. The implementation class name `XMLObjectStore` is stored in a configuration file. This file is stored under `META-INF/services` folder. The method `ServiceLoader.load(ObjectStore.class)` looks for the implementation classes inside the configuration file and returns the list to the service loader. In our example, we have only one implementation, therefore, we use the `loader.iterator().next()` method to retrieve the only implementation and assign it to the `store` field. There are several ways to register implementation classes with service locator such as a setter method can be used to assign any implementation of type `ObjectStore` (setter injection). Other ways to provide the implementation type include Java reflection and service registry pattern.

```java
public class BankingApp { ...
   private ObjectStore store;
   public void backup( ) {
      ServiceLoader<ObjectStore> loader = ServiceLoader.load(
         ObjectStore.class);
      store = (ObjectStore) loader.iterator().next;
      ...
   } ...
}
```

Listing 3.7: Service Locator Example

A disadvantage of using the service locator is that it leaves the client application unaware of new implementations registered with the service locator. For example, in the above example if a new and improved `ObjectStore` implementation is added to the service locator, the `BankingApp` class is not automatically notified about the new implementation. The `BankingApp` class has to make an explicit request to the service locator to obtain an implementation of `ObjectStore` as shown in listing 3.7. With dependency injection the `BankingApp` class does not have to make an explicit request, the service appears in the client class as shown in listing 3.6.

3.3.5 Type Generalisation

The declaration of variables with abstract types is considered good programming practice (Fowler, 1999). However, we find that in practice abstract types are rarely used to declare variables and fields in a class. For example, Steimann et al. (2006) presented a study, which showed that in several large Java projects only 1 out of 4 variables was declared through its interface. The type generalisation refactoring can be used along with the service locator pattern to decouple classes.

Example

Listing 3.8 shows an example where the declaration type of the variable list is abstracted from `ArrayList` to `List` interface. The type generalisation refactoring is only possible if the members invoked on that variable are part of the interface of the supertype.

```java
//Before refactoring
public class A { ...
   public void m(ArrayList list) {
      Object item = list.get(0);
      ...
   } ...
}
//After refactoring
public class A { ...
   public void m(List list) {
      Object item = list.get(0);
      ...
   } ...
}
```

Listing 3.8: Type Generalisation Example

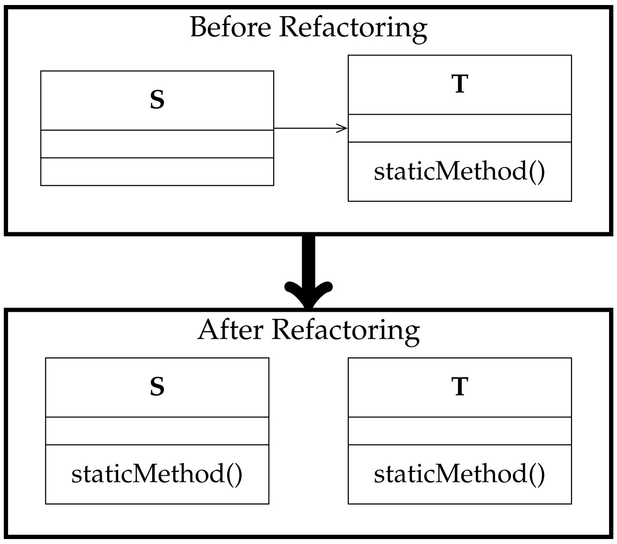
3.3.6 Static Members Inlining

In the source code of programs, some dependencies occur due to invocation of static members[6] of the target class. In some cases, static methods are used to declare utility functions that are self-contained, i.e., they do not use a class' data or methods. Therefore, to break a critical dependency, a possible solution is to inline the target static member into the source class. On the other hand, non-static inlining is much more complicated because non-static methods tend to use a class' data and methods.

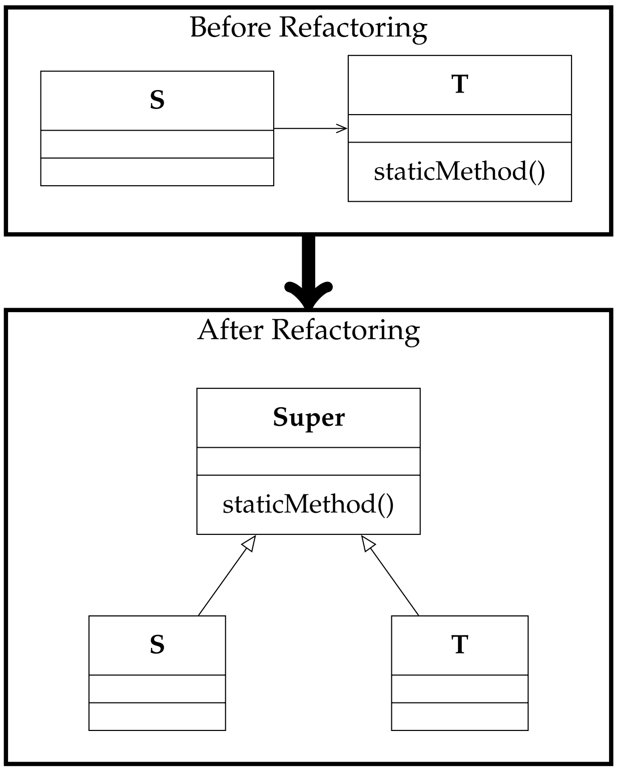
Inlining can be performed in four different ways:

1. **Copy Member**: In this scenario, the target static member is copied to the source class. This is not an invasive technique, however it would create redundancy in the code, which violates the DRY principle.

[6]We refer to public methods and fields as members of the class.

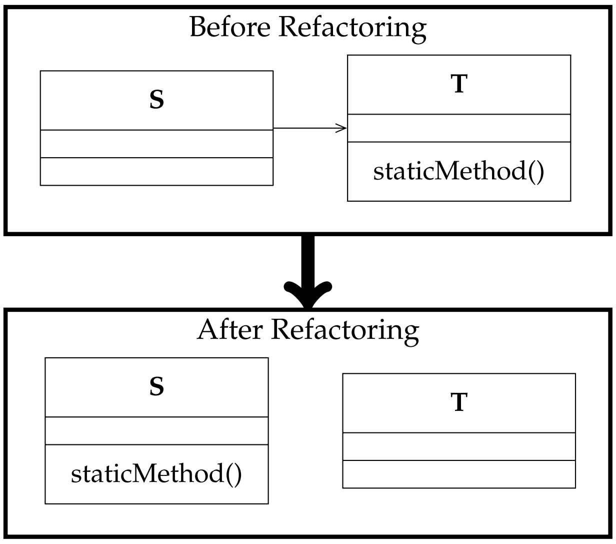


2. **Copy and Extract Member to Superclass**: To avoid duplication, a possible solution is to extract the duplicated code in a superclass, which is then extended by the source and the target class. However, this technique requires further refactorings such as introducing a new class, pulling up methods, modifying the source and the target class.

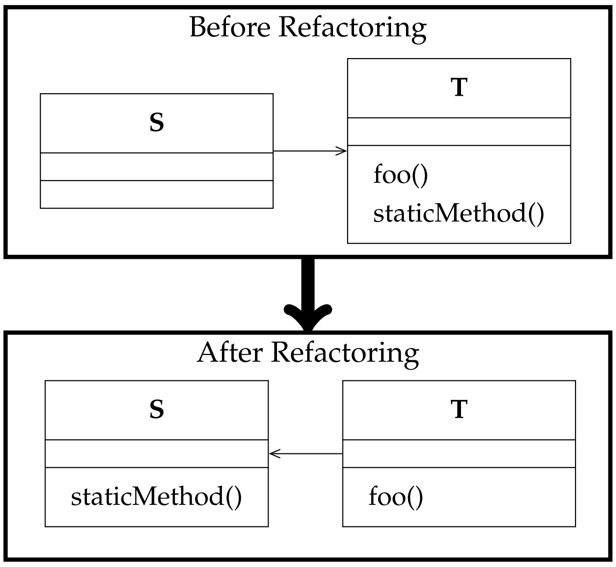


3. **Move Member**: In this case, the static member is moved from the target class to the source class. This is invasive refactoring as all clients of the static member

must be updated to refer to the new method location (i.e., the source class). This may potentially break code in libraries outside the scope of the program that is being refactored.

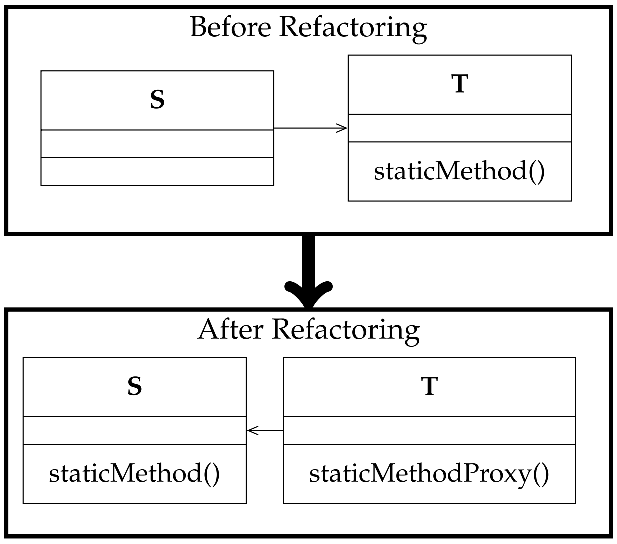


This refactoring may potentially reverse the dependency if `staticMethod()` is used by another method `foo()` in the target class.



4. **Proxy Method**: In this case, the move refactoring is used to move the referenced static member to the source class, but a delegate to the moved member is created in the source class. This avoids copying of code and the dependency between the source and the target class is inverted. This is not an invasive technique, as it

does not require updating clients of the static member. In the figure given below, `staticMethodProxy`[7] represents the delegate method.



Any of the above mentioned four types of refactorings can be recursive, i.e., when a method is either moved or copied, all its dependent members inside the target class must be moved or copied. Similarly, moving or copying a static field could also be recursive due to initialisation of the field. For example, the following code requires to move or copy the field and its dependencies: `public static Factory instance = createInstance()`. While applying static members inlining refactoring, we use the proxy method refactoring.

Example

Listing 3.9 shows a simple example of static method inlining. In this example the static method `printMessage` is moved into the source class S and the invocation statement is modified to refer to the new location. A delegate or proxy method is created in the target class, which now calls the static method in the class S.

```
//Before refactoring
public class S { ...
   public void m( ) {
      T.printMessage(message );
      ...
   } ...
```

[7]The delegate or proxy method may have the original name.

```
}
pubic class T { ...
   public static void printMessage(String m) {
      System.out.println(m);
   }
}
//After refactoring
public class S { ...
   public void m( ) {
      printMessage(message );
      ...
   }
   public static void printMessage(String m) {
      System.out.println(m);
   }  ...
}
pubic class T { ...
   public static void printMessage(String m) {
      S.printMessage(m);
   }
}
```

Listing 3.9: Static Member Inlining Example

3.4 Evaluation of Refactorings

There are two levels of refactorings: low-level or primitive and high-level or composite. The primitive refactorings are atomic refactorings that are used to perform local code level changes, for example, Fowler (1999) collected a catalogue of primitive refactorings: extract method, rename, move method, pull-up method, extract interface etc. These refactorings are readily available as part of modern IDEs, such as Eclipse, IntelliJ IDEA, and Netbeans. On the other hand, composite refactorings are a combination of primitive refactorings performed together to achieve large and complex refactorings (Opdyke, 1992). The dependency injection refactoring is an example of a composite refactoring, where we need several primitive refactorings to complete the task.

In the previous sections, we discussed several refactorings that can be used to break dependencies between classes. In our project, we aim to break critical dependencies between classes, therefore, the question arises as to which refactorings should be used for this purpose. In order to answer the question, we postulate several requirements based on refactoring properties to help choose a suitable set of refactorings for experiments.

Requirements

1. **Tool Support**: It is desirable to have existing tool support for a refactoring (such as move refactoring), because this provides confidence in the safe execution of the refactoring. We aim to fully automate the execution of these refactorings on the source code level. Therefore, a refactoring that is already available as part of an IDE would provide a solid foundation for the automation process.

2. **Conservativeness**: The primitive refactorings can be further divided into three categories: create, modify and delete. The create category includes refactorings that introduce new software artefacts, such as creating new methods, classes, or packages in the existing system. The modify category contains refactorings that alter existing artefacts. For example, renaming elements, changing accessibility of elements, pull-up methods, push-down methods, moving fields, methods or classes. The delete refactoring removes program artefacts, such as classes and package etc. We refer to the modify category as conservative refactorings because they do not remove or introduce new software artefacts whereas we call create and delete categories non-conservative refactorings. We choose to be conservative because we did not want to inflate an already inflated software. The introduction of new software artefacts comes with its own problems, for example, the name selection and placement of new artefacts. The removal or deletion of artefacts may affect the understandability of underlying system from developers perspective.

3. **Invasiveness**: The refactoring should not be invasive. We quantify the invasiveness property by the number of primitive refactorings required to perform a composite refactoring, which includes renaming a class, introducing a package, creating or moving a method etc. The number of refactorings do not provide an accurate measure as these are general suggestion provided by different sources. However, they give us an indication of the amount of change required in the structure of an existing program by a refactoring.

4. **Redundancy** The refactoring should not create redundancy in the program. The redundancy in software artefacts violates the DRY (Don't Repeat Yourself) principle and increases maintenance efforts. For instance, a refactoring should not generate duplicated code by copying methods in the program.

5. **Ripple Effects**: A refactoring may have ripple effects. Although tool support provides assistance in performing tasks across the whole program, it is desirable to minimise the impact of change in programs.

Evaluation

In this section, we evaluate the previously mentioned refactoring techniques according to the requirements. Our aim is to choose a refactoring set, which can be used to remove most of the dependency categories as discussed in section 2.5. Table 3.1 shows a comparison of different refactorings based on the requirements.

Refactoring	Tool Support	Conserv- ative	No. of Re- factorings	Avoids Re- dundancy	Ripple Effects
Adapt Parameter	NO	NO	6	YES	NO
Extract Interface	YES	NO	4	YES	YES
Constructor Injection	NO	NO	4	YES	NO
Setter Injection	NO	NO	4	YES	NO
Interface Injection	NO	NO	6	YES	NO
Service Locator	YES	NO	4	YES	NO
Type Generalisation	YES	YES	2	YES	YES
Static Members Inlining	YES	YES	3	NO	YES
Move Class	YES	YES	8	YES	YES
Split Package	NO	NO	13	YES	YES
Merge Package	NO	NO	13	YES	YES

Table 3.1: Refactoring Attributes in terms of Breaking Dependencies

Evaluating Tool Support: The tool support for some of the refactorings is provided by modern IDEs, such as Eclipse, Netbeans, and IntelliJ IDEA. The move class refactoring is part of all three IDEs. The static members inlining refactoring can be applied via the inline method or move method refactorings. These refactorings are also part of the previously mentioned IDEs. The type generalisation refactoring is known as *Use super type where possible*. This refactoring is available in Eclipse and Netbeans. The introduce factory refactoring can be used to partially implement a service locator. This refactoring replaces the object creation with a factory. This refactoring is provided by Eclipse and Netbeans. Similarly, the extract interface refactoring is also provided by all three IDEs.

Evaluating Conservativeness: The only refactorings that do not introduce new artefacts are: type generalisation, static members inlining and move class. These refactorings make use of existing artefacts to perform the desired action, for instance, moving methods or classes, and replacing types with their supertypes.

Evaluating Invasiveness: The information regarding the number of primitive refactorings is extracted from the following sources: adapt parameter (Feathers, 2004), extract interface (Fowler, 1999), the dependency injection and the service locator (Fowler, 2004), type generalisation; by counting basic steps required to apply the refactoring in the Eclipse IDE, static members inlining (Feathers, 2004), move class and split package (RefactoringCatalog, 1999), and merge package requires similar number of primitive refactorings as that of split package refactoring.

As mentioned before, the number of primitive refactorings may not reflect a clear picture and therefore we need to apply a subjective judgement in choosing the refactoring set. For example, Fowler (2004) defines the following base refactorings for introducing a setter injection: create a setter method, create the Assembler class, create the configuration method to set the Assembler class and create a test method for verification. Similarly, a service locator pattern is introduced by creating a global factory class, creating a method that returns the required information, adding a call to the new factory method, and creating a test method for verification. In both cases, four primitive refactorings are required, however, if we further analyse, we find out that in setter injection we need to introduce a new setter method in the source class for every dependency we try to break, while this is not the case in service locator pattern, i.e., only a statement is modified in the source class.

Evaluating Redundancy: The static members inlining refactoring may cause redundancy in the code. This happens when a static method is copied to the client class in order to break the dependency, creating code duplication. In order to avoid duplication, we use the move method refactoring to move the static method to the client class while leaving a delegate or proxy method in the original class.

Evaluating Ripple Effects: Some refactorings have ripple effects, for instance, when a new interface is extracted from existing types, we need to replace existing variable declarations with the new interface. The type generalisation refactoring also requires recursive modification in method signatures of subtypes, if the return type of a method in a supertype is generalised. In a similar way, the move refactoring may require renaming a class or changing the accessibility of different elements. In the example given below, we cannot move the class A outside the package p1 without changing the visibility of the method `m` in both classes. Such refactorings are known as composite refactorings, i.e., they are combination of smaller refactorings.

```java
package p1;
public abstract class A {
   abstract void m ();
}

package p1;
public class SubA {
    ...
    @Override
    void m() {
        ...
    }
}
```

Listing 3.10: Example of recursive changes required before moving the class A

Final Selection

On the basis of our requirements and evaluation criteria, we have chosen the following refactorings:

1. Service Locator
2. Type Generalisation
3. Static Member Inlining
4. Move Class

We use a combination of these refactorings to deal with nine dependency categories discussed in section 2.5. For example, when we encounter *implements*, *extends*, or *Other* dependency relationship, we apply the move class refactoring. Further discussion on how these refactorings are applied can be found in section 5.4.1.

Chapter 4

Applying Package Level Refactorings

In this and the next chapter, we aim to follow the methodology (Section 2.4) to identify critical dependencies and break those dependencies using package and class level refactorings (Chapter 3). This chapter starts with applying the package level refactoring (move refactoring) to break dependency cycles between packages. The move refactoring is used to break cycles between packages, therefore, we use the count of SCD antipattern instances as the major fitness function for our analysis. In this chapter, we discuss algorithms and tools used for the analysis of programs in the dataset. We also define pre and postconditions for the move refactoring. In order to validate our approach, we have developed an Eclipse plugin, which detects the move class refactoring on the model level and applies it to the source code. Finally, we discuss results obtained from the analysis.

4.1 Overview

We have used the move class refactoring to break dependency cycles between packages. This includes identifying misplaced classes in a system and moving them to more appropriate packages. The move class refactoring can be completely simulated on the model level (the dependency graph). The simulation provides the ability to evaluate the impact of refactoring without changing the source code of programs.

While the move class refactoring itself is rather simple and robust tool support exists to automate it, it is less clear how to find the right classes to be moved. We have developed an algorithm to identify classes that could be moved to other packages. This algorithm initially simulates a refactoring on the graph level and then applies it to the sources. The algorithm computes all paths in the dependency graph of a program that cause circular dependencies between packages. Once this is done, a scoring function ranks edges according to their number of occurrences in those paths. This ranking provides a good starting point to consider high-scoring edges as refactoring opportunities.

The algorithm starts with a high-scoring edge and checks pre and postconditions for each of the two candidate vertices (classes) attached to the edge. We have defined pre and postconditions on graph and code level. This means that we check graph level conditions and then we check conditions on code level. That is, given a class (vertex), all preconditions are checked and the move refactoring is simulated on the graph level. Next, postconditions are checked to ensure that this refactoring is suitable to be performed on the code level. If all graph level conditions pass for both candidate vertices of the edge, the algorithm chooses a class to move based on a selection criteria. The selection criteria requires that a move class refactoring should remove the highest number of circular dependency instances. On the contrary, if none of the two candidate classes pass all conditions, the algorithm continues with the next high-scoring edge. This is an iterative process, the algorithm keeps trying high-scoring edges in descending order until it reaches an edge where all pre and postconditions are satisfied for either of the candidate classes.

Once a refactoring is identified and simulated on the dependency graph level, this refactoring is applied on the source code level, i.e., we change the package declaration in the respective compilation units of the program. After performing the move refactoring, postconditions on the code level are checked to ensure that the refactoring didn't introduce any errors. If this is the case, we proceed with the identification of next refactoring. This process continues until all cycles are removed from the program or a certain number of refactorings are performed. It turns out that our approach does not, in general, result in a trivial solution where cycles disappear because packages are merged. This is discussed in detail later in section 4.6.6.

The evaluation of the proposed methodology is performed on a set of 92 open source systems collected from the Qualitas Corpus (Tempero et al., 2010). The key finding is that all the programs having circular dependency instances were improved in terms of a decrease in the total number of antipattern instances. Our approach improves other standard metrics related to strongly connected components. The main strength of our approach is that the analysis is initially performed on the dependency graph, which reduces the chances of introducing errors in the programs, i.e., we can predict to a certain extent that the refactoring would not break the system. In order to apply identified refactorings on the source code level, we have developed an Eclipse plugin, which is an extension of the standard move class refactoring built in to Eclipse, but it is fully automated to perform batch operations on sets of programs. Further discussion about the plugin can be found in section 4.4.

4.2 Background

The move class refactoring has been used to re-modularise programs, improve modularisation existing programs, and remove antipatterns. For example, Seng et al. (2005) have developed an algorithm to re-modularise programs by improving subsystem decomposition. The authors have used a fitness function based on coupling, cohesion, complexity, cycles and bottlenecks. They treated the subsystem decomposition as a search problem and used a genetic algorithm to identify the move class refactoring opportunities. Their results show improvement in other metrics except the coupling and cohesion of individual packages. Their work is related to software re-modularisation as their solution generates new packages and suggests classes to move there. On the other hand, we focus on improving the existing structure of programs.

Hautus (2002) has suggested PAckage STructure Analysis (PASTA) metric to measure the degree of cycles between packages in a program dependency graph. This metric can be used to identify undesirable dependencies between packages in a program. When these dependencies are removed, the package structure would become acyclic. The author has suggested the move class refactorings as a solution to get rid of package cycles. However, his approach does not identify classes that should be moved to other packages in order to remove cycles. O'Keeffe and O'Cinneide (2006) have developed a metric suite to form a fitness function, which they use to identify refactoring opportunities and improve software quality. They use the QMOOD quality model (Bansiya and Davis, 2002) to evaluate alternative solutions. The authors use search techniques such as Hill Climbing and Simulated Annealing to explore alternative solutions in the refactoring process. Refactorings supported by their methodology are limited to the hierarchical structure of design, which include extracting and collapsing hierarchies, moving features up and down the hierarchy.

Abdeen et al. (2009) have looked at the modularisation problem of existing systems from the perspective of object-oriented design principles, such as acyclic dependency principle, closure change principle, and closure reuse principle. They have used simulated annealing techniques to optimise the package structure of a program by suggesting move class refactorings across different packages. In their approach, they allow maintainers to specify filters (package size, maximum number of classes to move) before performing move refactorings. A drawback in the approach is that it doesn't bring class visibility into account, which can be an important factor while moving classes on the source code level. In addition, the authors have performed experiments on the dependency graphs of programs and they do not suggest how to translate their algorithm on the source code level. Laval et al. (2009) have used an enriched dependency structure matrix (eDSM) to visualise dependencies among software artefacts. The visualisation highlights any cyclic dependencies between packages, and suggests move class refactorings. However, there is no implementation of their approach to automatically apply those refactorings on the code level.

In the literature, clustering techniques have been used to restructure legacy systems (Maqbool and Babri, 2007; Mitchell and Mancoridis, 2006). Czibula and Serban (2007) have proposed the use of a hierarchical agglomerative clustering algorithm to recondition system classes. Their methodology includes: extracting data (classes, methods, attributes and their relationships), re-grouping the extracted data using their clustering technique and extracting refactoring suggestions by comparing the old and the modified software structures. The refactorings proposed by their algorithm are Move Method, Move Attribute, Inline Class and Extract Class. A major disadvantage of their technique is that it provides solutions in terms of aggregation of refactorings, which means designers have to either accept or reject the solution in its entirety. On the other hand, an iterative approach to refactoring leads to a gradual system change helping designers to asses the impact of refactorings at each iteration.

4.3 Algorithm

We have developed a novel algorithm, which automates the refactoring process. This process includes identifying move refactorings on the model level, checking refactoring constraints, and applying those refactorings on the code level. The algorithm is greedy, i.e., it does not perform backtracking. The reason we chose greedy algorithm over other techniques is mainly due to performance. Table 4.8 in the results section shows the execution time of some of the large programs in the dataset. The use of backtracking would have dramatically increased the execution time. Algorithms 1, 2 and 3 show the main steps of the algorithm.

Algorithm 1 executeMoveClass

```
Input: graph, pres, posts
Output: a list of refactorings
 1: {Define a list, which holds objects of type move class refactoring}
 2: refactoring list ← new List
 3: iteration ← 1
 4: MAX-ITERATIONS ← 50
 5: while iteration ≤ MAX-ITERATIONS do
 6:    instances ← computeSCDInstances(graph)
 7:    if |instances| = 0 then
 8:       return list
 9:    end if
10:    highScoredEdges ← getHighScoredEdges(insances)
11:    for highScoredEdge in highScoredEdges do
12:       succeeded ← applyMove(graph, highScoredEdge, instances, pres, posts)
13:       if succeeded then
14:          list.add(refactoring)
15:          graph ← buildGraph()
16:          iteration ← iteration + 1
17:          break
18:       end if
19:    end for
20: end while
21: return list
```

Algorithm 2 applyMove

```
Input: graph, highScoredEdge, pres, posts
Output: true or false
 1: candidate ← getMoveCandidate(graph, highScoredEdge, pres, posts)
 2: if candidate ≠ null then
 3:    succeeded ← applyOnCode(candidate, pres, posts)
 4:    if not succeeded then
 5:       rollback(candidate)
 6:       {Try the reserved candidate}
 7:       if candidate.getReservedCandidate() ≠ null then
 8:          succeeded ← applyOnCode(candidate.getReservedCandidate(), pres,
            posts)
 9:          return succeeded
10:       else
11:          return false
12:       end if
13:    else
14:       return true
15:    end if
16: else
17:    return false
18: end if
```

Algorithm 3 getMoveCandidate

```
Input: graph, highScoredEdge, pres, posts
Output: candidate or null
 1: source ← highScoredEdge.getStart()
 2: target ← highScoredEdge.getEnd()
 3: candidate1 ← checkAllConditions(source, graph, pres, posts)
 4: candidate2 ← checkAllConditions(target, graph, pres, posts)
 5: if candidate1.isPassed() and candidate2.isPassed() then
 6:    {Choose the one which removes the most number of instances}
 7:    if candidate1.instanceCount() ≤ candidate2.instanceCount() then
 8:       candidate1.setReservedCandidate(candidate2)
 9:       return candidate1
10:    else
11:       candidate2.setReservedCandidate(candidate1)
12:       return candidate2
13:    end if
14: else if candidate1.isPassed() and not candidate2.isPassed() then
15:    return candidate1
16: else if not candidate1.isPassed() and candidate2.isPassed() then
17:    return candidate2
18: else
19:    return null
20: end if
```

4.3.1 Building the Dependency Graph

We build the dependency graph of a program from the bytecode. The dependency graph comprises of vertices (classes) and edges (relationships between classes). The process of extracting the dependency graph is explained in section 2.1. Algorithm 1 uses this graph as an input to compute and perform refactorings.

4.3.2 Computing Antipattern Instances

In this step, we compute the total number of SCD instances. The Guery tool (version 1.3.5) is used to compute antipattern instances, as discussed in section 2.2.2. We keep track of strong (SCD) and weak (WCD) circular dependencies between packages. The detailed discussion about antipatterns can be found in section 2.3. The function `computeSCDInstances(graph)` (Algorithm 1, Line 6) is used to compute the total number of instances present in the program. We use the total number of SCD instances as a metric to compute the impact of a move refactoring, i.e., a refactoring must decrease the total number of instances.

4.3.3 Computing Edge Scoring

The next step involves the computation of high-scored edges representing potential refactoring opportunities. The process of computing high-scored edges is explained in section 2.4. The funtion `getHighScoredEdges(instances)` (Algorithm 1, Line 10) returns a list of edges sorted in descending order by their score value.

We are only interested in edges with a score greater than zero and they must transition between packages. We define this as follows. Let `G = (V, E)` be the dependency graph where `V` be the set of all classes in a program and `E` be the set of all edges in `G`. Let `score(e)` represent a natural number associated with an edge such that $e \in E$, and let `namespace(e.<end>)` represent the package name of $< end >$, where $< end >$ is either source or target.

$$\forall e \in E \mid score(e) > 0 \wedge namespace(e.source) \neq namespace(e.target)$$

The algorithm then iterates over all the top scored edges (Algorithm 1, Line 11) until it finds an edge where all pre and postconditions for the move refactoring are satisfied.

The algorithm can be implemented as anytime algorithm (Zilberstein, 1996). The computation of edge scoring could be interrupted so that it computes score under a certain threshold instead of all antipattern instances. This would still give us approximate bad edges (critical dependencies). This means we could get meaningful results even if the algorithm does not hit the optimum.

4.4 Implementation: CARE - The Eclipse Plugin

The aforementioned algorithm is implemented as an Eclipse plugin called CARE (Code and Architectural Refactoring Environment)[1]. This plugin is an extension to the move refactoring built into Eclipse. The purpose of this plugin is to automate the process of computing and executing move class refactorings in a program. The Eclipse IDE provides support for an individual Move refactoring, but there is no support for performing this refactoring in an automated way in a batch setting. We load all programs as Eclipse projects in a workspace and with a push of a button, the plugin iterates over all projects to perform refactorings.

[1]Appendix B describes how the plugin can be installed and used

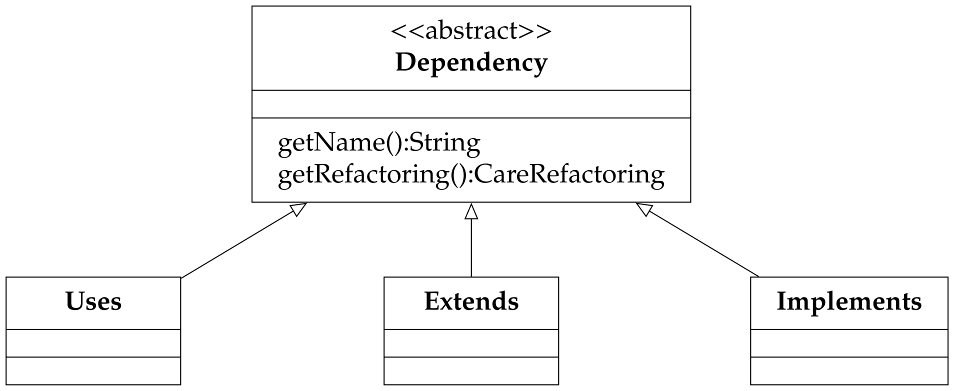


Figure 4.1: Class Diagram of Dependency Classification for Move Refactoring

4.4.1 Implementing Dependency Classification

We have discussed several categories of a dependency relationship in section 2.5. Three major categories are as follows: *uses*, *extends*, and *implements*. The class diagram of dependency classification is shown in figure 4.1. The abstract class `Dependency` has two methods: `getName()` - returns the name of dependency and `getRefactoring()` - returns the respective refactoring to be applied. In this experiment, the default refactoring is the move class refactoring for all dependency categories.

4.4.2 Implementing Refactoring Constraints

Refactoring constraints refer to pre or postconditions that are checked before and after a refactoring is performed. The constraints mechanism is part of a refactoring meta-model which is implemented directly in Java as part of our tooling. The class diagram of constraints is shown in figure 4.2. Each constraint has three methods: `check`, `getName`, and `isGraphLevel`. The `check` method returns true if a preconditions is fulfilled and false otherwise. The `getName` method returns the name of a precondition and `isGraphLevel` determines whether the precondition can be checked on the graph or code level.

Preconditions

There are three preconditions that are checked before the move refactoring is performed.

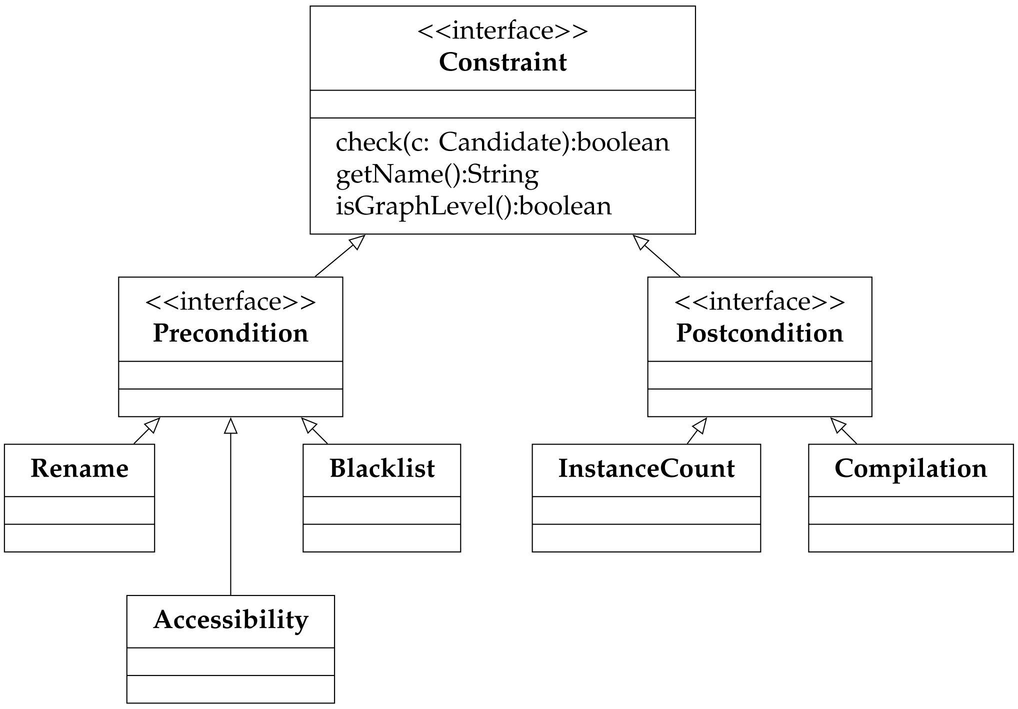


Figure 4.2: Class Diagram of Pre and Postconditions

4.4.2.1 Rename Precondition

When we move a class from one package to another, it is important to check that the target package should not contain a class with the same name as that of the class to be moved. If a class with the same name exists, the move refactoring is skipped. This precondition is checked on the graph level.

Let C be the set of all classes in a package P. Let name(classToMove) be the name of class to be moved to the package P. Then we define the rename precondition as follows:

$$\nexists\, c \in C \mid name(classToMove) = name(c)$$

4.4.2.2 Blacklist Precondition

Our algorithm automatically performs the move refactoring in a program. However, in some cases a developer may want that certain classes should not be moved. In that case, we can manually specify a list of fully qualified class names in the Eclipse preferences section of CARE plugin. This is a graph level precondition. This precondition checks whether a class to be moved to another package is blacklisted or not. If a class' name exists in the blacklisted classes, the move refactoring is not performed.

Let B be the set of blacklisted classes and let name(classToMove) be the name of the class to be moved to another package. The blacklist precondition is defined as follows:

$$\nexists\, b \in B \mid name(classToMove) = name(b)$$

4.4.2.3 Accessibility Precondition

Accessibility modifiers establish rules for information hiding within Java programs. There are four types of accessibility modifiers in Java: public, private, protected, and default (no modifier). These modifiers can be applied on two levels:

- Top level (classes): public or default.
- Member level (fields, methods, inner classes): public, private, protected, and default.

The access level of a class' artefacts determines whether other classes can access fields or methods of this class or not. The purpose of these modifiers is to support encapsulation.

The accessibility precondition checks whether a class to be moved requires changes in the accessibility level of this class or other classes. Consider the example in listing 4.1. In this example, if we try to move the class `A` outside of the package `a`, we need five change accessibility refactorings. This includes changing the access modifier of class `A` and `C` to public. We also need to change the access modifiers of the field `A.cObject`, the method `A.m()` and `C.m()` to public.

We used the Dependency Finder (Tessier, 2010) to extract the accessibility information of a class. For every move candidate class, we compute all non-public incoming and outgoing references of the class within the original package. In listing 4.1 the class `A` has three incoming references (from class `B`) at the class, field and method level. There are two outgoing references from the class `A` to the class `C`. If the move class refactoring requires changing accessibility modifiers, we skip the refactoring to preserve encapsulation.

Let X be the set of all move candidate classes and let countAccessabilityChanges(class) determine the number of change accessibility refactorings required in order to move the class. Then we define the accessibility precondition as follows:

$$\forall\, c \in X \mid countAccessabilityChanges(c) = 0$$

```
package a;
class A {
   protected C cObject = new C();
   void m() {
      cObject.m()
   }
}
package a;
class B {
   void m() {
      A a = new A ();
      a.m();
      a.cObject.m();
   }
}
package a;
class C {
   void m() { }
}
```

Listing 4.1: Change Accessibility Example

Postconditions

There are two postconditions for the move class refactoring.

4.4.2.4 Instance Count Postcondition

There could be a situation where a move class refactoring may increase the number of instances as compared to the number of instances before performing the refactoring. Figure 4.3 shows an example where the number of instances increase after a refactoring. In this figure, two edges (C3→C1 and C1→C2) have the same score of one. When two edges have the same score, they are sorted in ascending order according to the fully qualified names of start and end vertex of each edge. Therefore, the edge C3→C1 is picked as the first high-scored edge. In this case, C3 was moved from packagey to packagez. There was only one SCD instance before refactoring, while after refactoring the number of instances increased to two[2]. Table 4.1 shows paths that create circular dependencies before and after the refactoring.

[2]We have used find all attipattern instances option of the Guery engine to compute the instances.

Instance Type	Packages	Paths
SCD Before	packagey, packagez	C3, C1, C2
SCD After	packagey, packagez	C5, C3, C4
	packagey, packagez	C5, C3, C1, C2

Table 4.1: Instance Count Before and After Refactoring

If a situation arises where the number of instances increase after a refactoring, we avoid the move refactoring. This postcondition can be easily checked on the graph level, where we compare the instance count before and after the refactoring. Let g be the graph before the move refactoring and let $g\prime$ be the graph after refactoring. Then we define the instance count postcondition as:

$$instanceCount(g) \geq instanceCount(g\prime)$$

In the postcondition we perform a refactoring even if it doesn't remove any instances. In dense dependency graphs it is possible that one edge removal is rerouted through a different path to create another instance. Therefore, the total number of instances do not decrease, however it is possible that the newly created cycle is removed in the next iterations, which would eventually decrease the number of instances. Figure 4.4 shows an example of instance count postcondition. In this figure the total number of SCD instances is 1 before refactoring. After the first refactoring the number of instances remain the same. However, after the second refactoring the number of instances decrease to zero.

4.4.2.5 Compilation Postcondition

The move class refactoring should be safe, i.e. it should not introduce any compilation errors. However, there are some cases where the program compilation fails after performing the move refactoring. These situations are listed in the section 4.6.11. Therefore, once all other pre and postconditions are passed, this last postcondition (on code level) is checked to ensure that the move refactoring executed successfully.

4.4.3 Implementing Refactorings

In order to implement refactorings, we have used the Eclipse Language Toolkit (LTK). This toolkit is a refactoring framework and it provides access to the Eclipse IDE's code level refactorings, such as move refactoring, pull up / push down methods. The base

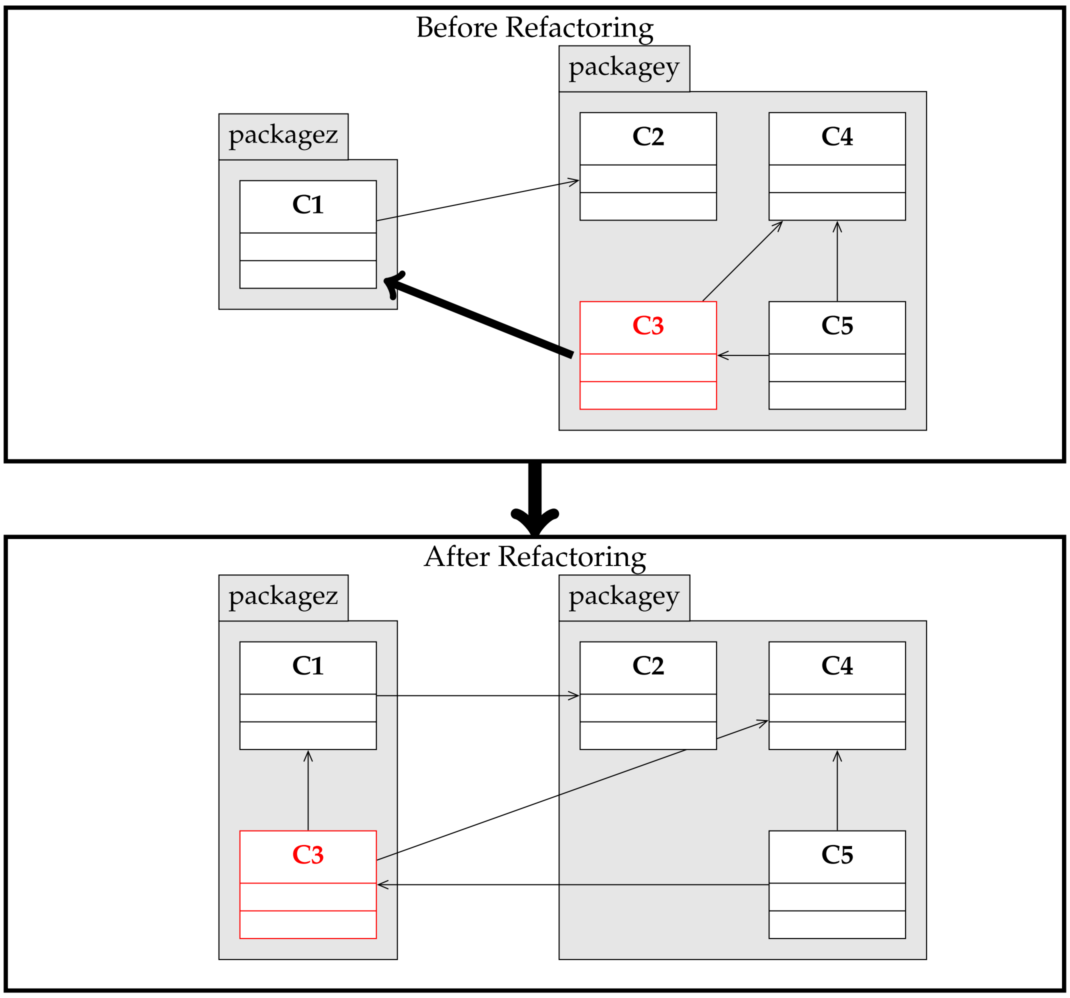


Figure 4.3: Example of Increase in the Instance Count Metric

class for all refactorings is `org.eclipse.ltk.core.refactoring.Refactoring`. This is an abstract class and all refactorings inherit methods from this class. The `Refactoring` class performs two actions: checks several conditions to make sure that the refactoring can be safely executed and creates a `Change` object which represents the source code modifications.

In our approach, we have extended the base `Refactoring` class and created another abstract class `CareRefactoring` as shown in figure 4.5. This class performs three major actions:

1. It checks a set of preconditions before applying a refactoring. It calls `checkAll-Conditions()` method from the superclass `Refactoring`.

2. If the preconditions succeed, refactorings are performed tentatively on the source

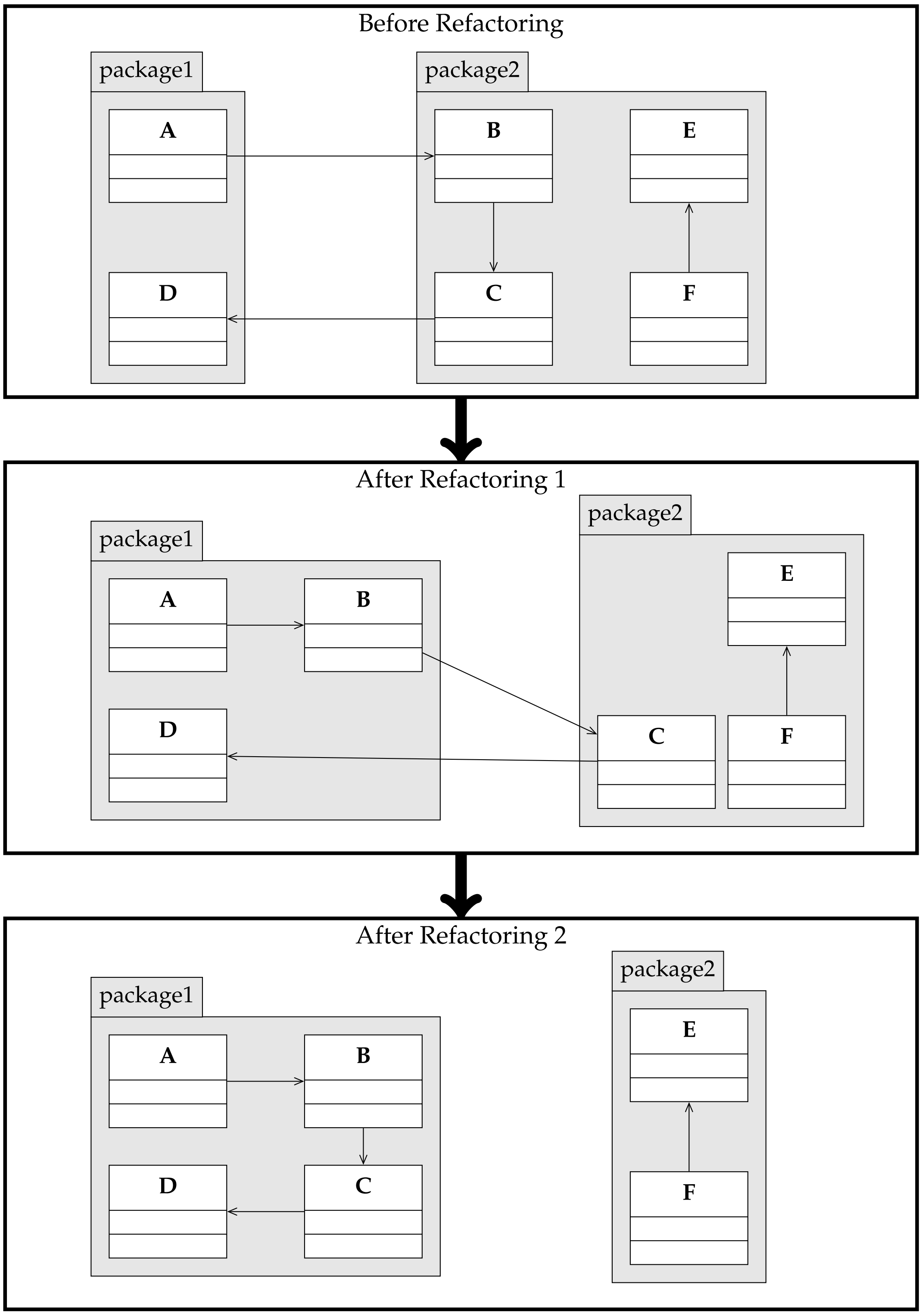


Figure 4.4: Example of Decrease in the Instance Count Metric

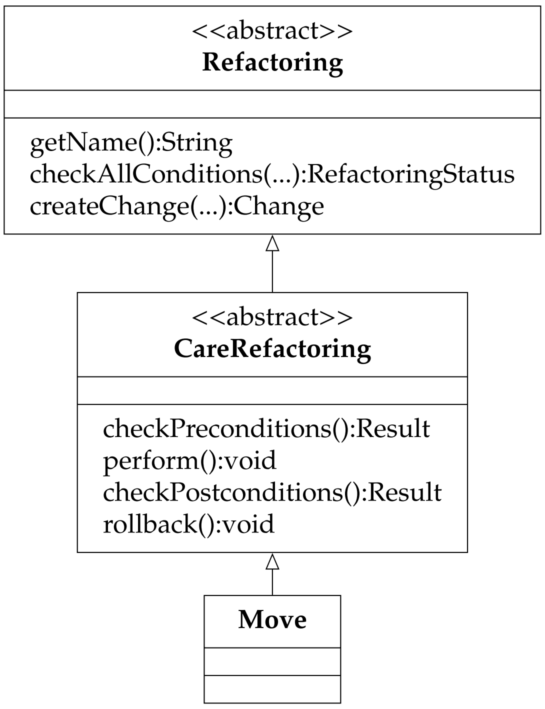


Figure 4.5: Class Diagram of Refactorings

code. This step involves executing the `createChange()` method of the `Refactoring` class.

3. Postconditions are checked to determine whether or not refactorings were performed successfully. If postconditions pass, the refactoring process is considered successful, otherwise the code level modifications are reversed using the `rollback()` method.

The `Move` refactoring is built on top of the standard Eclipse refactoring `MoveRefactoring`. In this refactoring a class is moved from one package to another. First, the algorithm performs move refactoring on the dependency graph to evaluate its impact. Next, it is executed on the code level, if all pre and postconditions succeed.

4.5 Strongly Connected Component Metrics Definition

We have used the instance count of strong circular dependency between packages as the major metric to compute the impact of refactorings on programs. However, there are several other metrics - related to the package structure - that can be used to gauge the improvement in the program structure. For example, package cycles can be looked at as strongly connected components (SCC). Within a SCC a vertex has a path to every other vertex. We have used different metrics to compute the impact of refactorings on strongly connected components in the dependency graph. By using these metrics we have tried to asses the complexity of SCCs before and after refactorings.

First, SCD and WCD counts are recomputed. Next, we look into various metrics related to strongly connected components (SCCs). Note that we are not considering the SCCs in the original dependency graph, but in the derived package dependency graph. In this graph packages are vertices and edges are the inferred relationships between packages as defined by (Falleri et al., 2011). In this package dependency graph, the first metric computed is the size of the largest SCC (MAX_SCC).

The minimum feedback arc (edge) set size is a measure indicating how many edges have to be removed in order to make the SCC acyclic. Unfortunately, computing the minimum feedback arc set size is a NP-hard problem and scalable algorithms are only available for approximations (Eades et al., 1993). We therefore decided to use two alternative metrics to measure the thickness of SCCs.

Let G = (V, E) be a directed graph of packages and their relationships and let $\{SCC_i\}$ be the enumerated set of all strongly connected components in G. We only considered components with at least two vertices to exclude trivial SCCs. For a component SCC_i; let V_i be the set of vertices in SCC_i and E_i be the set of edges in SCC_i.

The minimum possible number of edges $minE_i$ in SCC_i is $|V_i|$. This is exactly the case when the component consists of one simple cycle. The maximum possible number of edges $maxE_i$ in SCC_i is $|V_i| * (|V_i| - 1)$. This is the case when the component forms a clique.

Density

Density measures the density of edges within a component. We define the density of a SCC as a relative distance from the minimum possible number of edges scaled to the maximum possible number of edges within the component. In other terms, we use a simple min-max normalisation. The minimum value (0) means that the component has minimum possible number of edges and hence easy to break the tangle. On the flip side, the maximum value (1) means that the component has maximum possible edges making it difficult to break the tangle. The extreme value represents a clique and a cycle, as shown in figure 4.6.

$$D(SCC_i) := \frac{|E_i| - minE_i}{maxE_i - minE_i} = \frac{|E_i| \; - \; |V_i|}{|V_i|^2 - 2\,|V_i|}$$

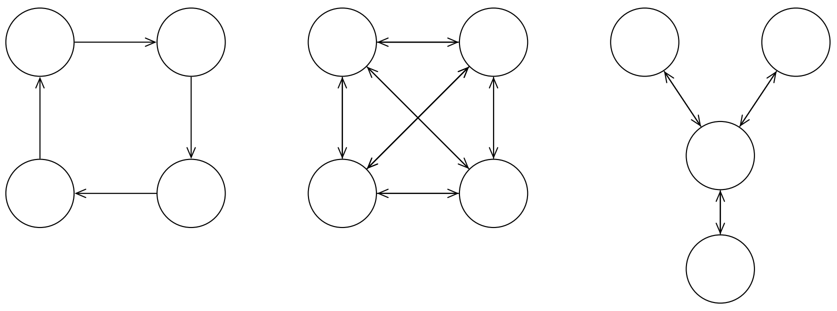


Figure 4.6: SCC-A (Left), SSC-B (Middle), SCC-C (Right)

Metric	SCC-A	SCC-B	SCC-C
Density	0	1	0.25
Tangledness	0	1	1

Table 4.2: Metric Values of Three SCCs

Tangledness

Tangledness measures the length of cycles within a component. Falleri et al. (2011) proposed a measurement to calculate the tangledness of a component. The idea is to measure the length of the shortest path representing a back reference for each edge in the SCC. i.e., for each edge e connecting vertices s and t, we are interested in the shortest path from t to s. By definition of a SCC such a path must exist within the SCC. The value of this metric is scaled between 0 and 1.

The maximum possible average length $maxSPL_i$ of the shortest paths is $|V_i| - 1$. This is exactly the case when a component consists of one simple cycle. The minimum possible average length $minSPL_i$ is one. This is the case when for each edge there exists a reverse edge in the SCC. Let SPL_i be the average length of the shortest path for all edges in SCC_i. In analogy to the density metric, we define tangledness as :

$$T(SCC_i) := 1 - \frac{SPL_i - minSPL_i}{maxSPL_i - minSPL_i} = 1 - \frac{SPL_i - 1}{|V_i| - 2}$$

Figure 4.6 shows examples of three different SCCs and their values are shown in table 4.2. In these SCC metrics smaller values are considered to be better.

No.	move type	to package
1	net.sf.jmoney.Start	net.sf.jmoney.gui
2	net.sf.jmoney.SortedTreeNode	net.sf.jmoney.model
3	net.sf.jmoney.Currency	net.sf.jmoney.model
4	net.sf.jmoney.Constants	net.sf.jmoney.model
5	net.sf.jmoney.io.QIF	net.sf.jmoney.gui
6	net.sf.jmoney.io.MT940	net.sf.jmoney.gui
7	net.sf.jmoney.SortedTreeModel	net.sf.jmoney.model

Table 4.3: The Resultant Move Refactorings for JMoney-0.4.4

4.6 Experiment

In order to validate our approach, we applied it to 92 open source Java programs. First, we discuss two case studies and then we explain the results obtained over 92 programs.

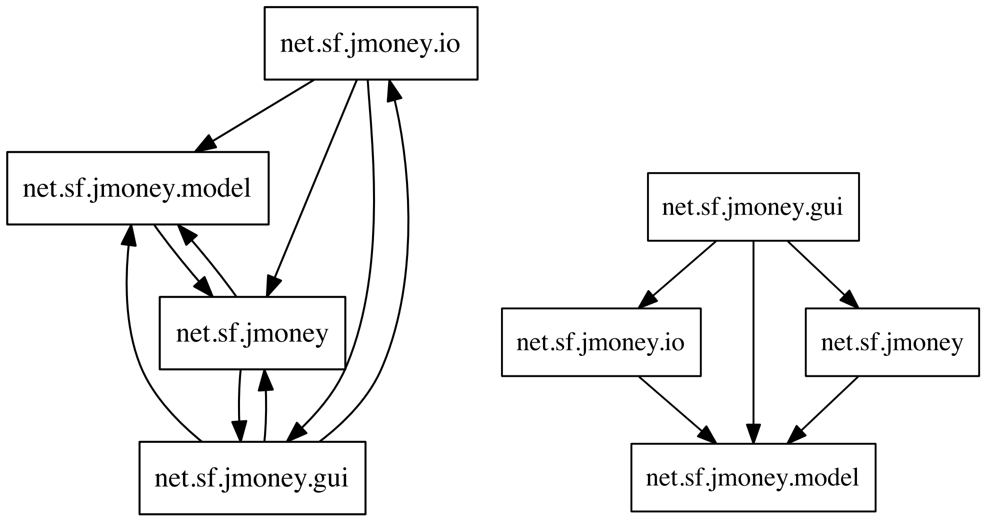
4.6.1 Case Study: JMoney-0.4.4

JMoney is a personal accounting software built using Eclipse RCP. The dependency graph of this program has 193 vertices and 548 edges. Our algorithm implementation computed and performed move refactorings for this program in 27 seconds[3]. After 7 move refactorings the total number of SCD and WCD instances dropped to zero. The suggested refactorings are shown in table 4.3.

In order to examine whether the proposed refactorings make sense to a developer, we have manually investigated the source code of the classes to be moved. For example, the class `net.sf.jmoney.Start` is an entry point to run the application. This class invokes `net.sf.jmoney.gui.MainFrame` to load the application's GUI. Therefore, this class is a good candidate for moving to the gui package. The class `net.sf.jmoney.-Currency` is a simple model to represent currencies. This is used by a class `net.sf.-jmoney.model.Account`, which is a data model for an account. In order to keep domain classes together, the `Currency` class should be moved to `net.sf.jmoney.model` package. Similarly, the other two classes `SortedTreeNode` and `SortedTreeModel` are also required by data model classes in the `net.sf.jmoney.model` package. Therefore, these two classes also moved to the model package.

The two classes in `net.sf.jmoney.io.QIF` and `net.sf.jmoney.io.MT940` provide the

[3]Macbook Pro Intel Core i7, 4GB RAM, JRE 6



(a) Before Refactoring **(b)** After Refactoring

Figure 4.7: Package Dependency Graph of JMoney-0.4.4

facility of importing and exporting files through JMoney GUI. These two classes have cyclic relationship with `net.sf.jmoney.gui.MainFrame`, which builds the application's GUI. Our algorithm recommended to move `QIF` and `MT940` classes into `net.sf.jmoney.gui` package. Another solution could be to break the `MainFrame` dependency from `QIF` and `MT940` classes. This involves a program's source code analysis and modification. Since our algorithm does not modify the program's structure, the first solution was suggested.

We have extracted a package dependency graph (PDG) from the original class dependency graph to visualise the package level tangles. Figure 4.7a shows the PDG before performing move refactorings, while figure 4.7b represents the PDG after performing the refactorings as shown in Table 4.3. Table 4.4 shows before and after values of different metrics defined in section 4.5. The results show a significant improvement in all metrics. Note that these refactorings do not result in a trivial solution where packages are emptied (see discussion in Section 4.6.6).

4.6.2 Case Study: JGraph-5.13.0

JGraph is an open source graph visualisation and layout library. The dependency graph of JGraph consists of 187 vertices and 797 edges. The algorithm computed 10 refactorings to completely untangle the program. It took 58 seconds[4] to compute and

[4]Macbook Pro Intel Core i7, 4GB RAM, JRE 6

Metric	Before	After
SCD Count	117	0
WCD Count	12	0
Package Count	4	4
Max SCC Size	4	1
Average Density (G)	0.64	0
Average Tangledness (T)	0.79	0

Table 4.4: Metrics Values for JMoney-0.4.4

Metric	Before	After
SCD Count	3475	0
WCD Count	30	0
Package Count	20	18
Max SCC Size	6	1
Average Density (G)	0.05	0
Average Tangledness (T)	0.08	0

Table 4.5: Metrics Values for JGraph-5.13.0

execute these 10 refactorings. Table 4.5 shows the impact of refactorings on different metrics values.

An important point to note here is that the total number of packages in the program reduced from 20 to 18. Two packages that merged with other packages were `org.jgraph` and `org.jgraph.plaf`. Each of these two packages contained only a single top level Java class namely `JGraph` and `GraphUI` respectively. These two classes are related to a graph's display area and its look and feel. References to these classes mainly come from the `org.jgraph.graph` package. The algorithm suggested to move these classes to `org.jgraph.graph` package. In this way, the total number of remaining packages was reduced by two.

One of the other identified refactorings suggested to move a class `org.jgraph.plaf.-basic.BasicGraphUI` to `org.jgraph.graph` package. This class is an extension of the *GraphUI* class and provides a graph's data structure. This class is clearly a good candidate for moving into the relevant package `org.jgraph.graph`, which has classes related to the graph model, graph cells, controllers and drivers.

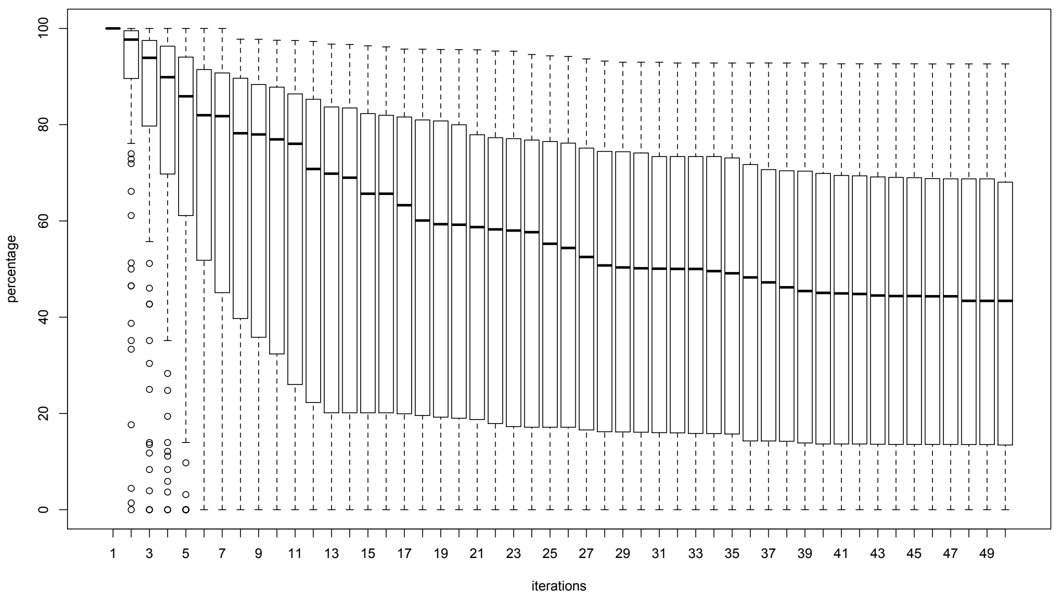


Figure 4.8: Decrease in SCD Instances After Move Refactorings

4.6.3 Impact of Move Class Refactoring

We have analysed 92 programs in the dataset. Out of those 92, 88 programs had SCD instances, for which we computed refactorings[5]. Figure 4.8 shows the decrease in SCD instances after every move refactoring. This is a boxplot chart where the box represents 50% of the observations. The horizontal line in the box represents the median. In this chart, we scale each observation by the number of original SCD instances for its corresponding program. The value on the x-axis represents the number of refactorings executed on the code level. Our experiment performs 50 refactorings on each program. We chose this number of refactoring because the decline in median value after 50 iterations flattens out to less than 1 percent. However, in the CARE plugin this value can be customised for individual projects.

It is notable that after 26 iterations the median value over all observations drops below 50%. This means, for more or less half of the programs, only 26 move refactorings are required to remove 50% of SCD instances. Similarly, only 7 refactorings are required to remove the first 20% of instances. The boxplot also shows some outliers. These are programs where the total number of SCD instances dropped to zero in a few initial steps.

In our experiment we kept a record of both SCD and WCD instances. Figure 4.9 shows the comparison of median values of SCD and WCD instances. This figure indicates that

[5]Programs with zero SCD instances are: fitjava-1.1, jfin-date-math-r1.0.1, jasml-0.10, javacc-5.0

the algorithm not only removes SCD instances, but as a by-product, it also removes WCD instances.

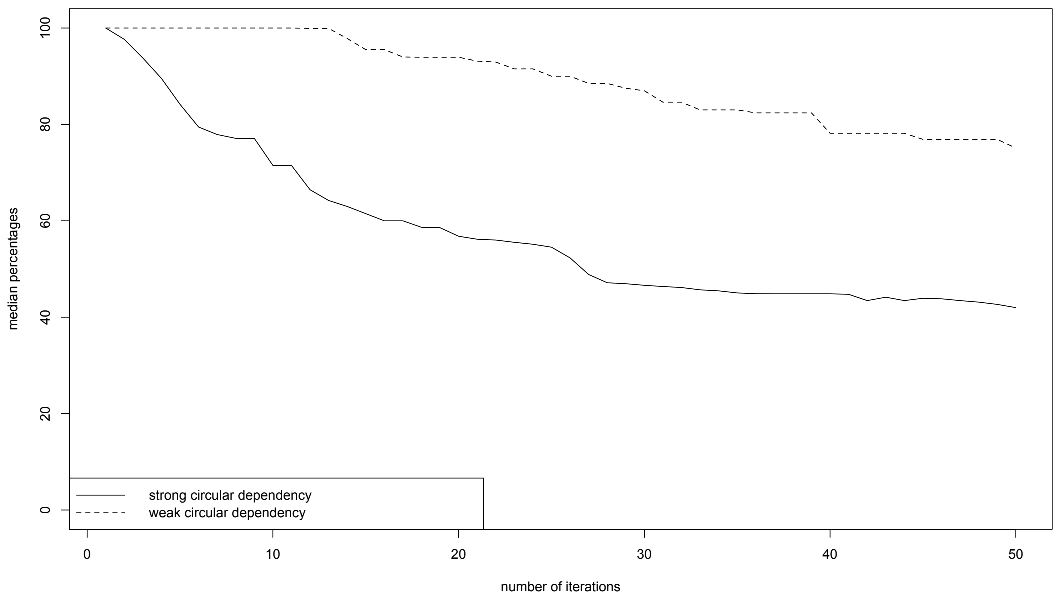


Figure 4.9: Decrease in no. of Instances: Comparison between SCD and WCD

4.6.4 Refactoring Simulation on Model vs Refactoring Application on Code

In this section, we compare the refactoring simulation on the graph level and the refactoring application on the code level. Figure 4.10 shows the comparison between the scenarios. The two lines represent the median value for each refactoring type.

There are 50 iterations and for every iteration, the simulation on the graph level is performed by removing the highest scored edge without checking any pre and postconditions as laid out in section 4.4.2. This is why we get a very steep curve representing the decline in the number of instances. However, not every edge can be removed on the code level as some pre or postconditions may fail. Consequently, our algorithm iterates over a list of high-scored edges until it reaches the one which passes all conditions. Therefore, the curve generated by the refactorings on the code level is less steep than on the graph level. The gap between the two scenarios can be further reduced by weakening the preconditions. For example, we may allow those refactorings that require further refactorings, such as changing accessibility and allowing rename. In that case the curve of code level refactorings will become closer to the one generated on the graph level.

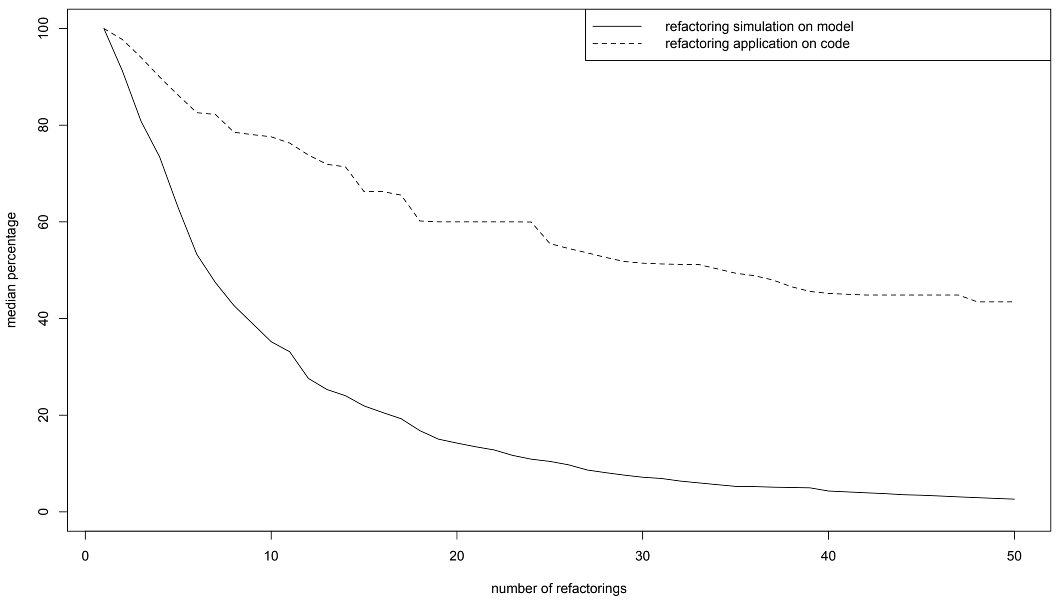


Figure 4.10: Decrease in no. of Instances: Comparison on Model and Code Levels

4.6.5 Impact of Program Size on Number of Refactorings

Figure 4.11 shows the number of refactorings that are necessary to remove 25% of antipattern instances, depending on program size measured by the number of vertices in the dependency graph. This figure shows that, for most programs, only few edge removals are necessary to achieve the removal of top 25% instances. However, there are a few programs that require a large number of edge removals. Programs with 50 refactorings were censored implying that 50 refactorings were not enough to remove 25% instances.

To quantify the relantionship between program size (vertex count) and number of refactorings, we applied a simple linear regression model on the number of refactorings to remove top 25% of instances in terms of vertex count. The fitted model is:

$$fitted\ number\ of\ refactorings = 8.98 + 0.0098 * vertexcount$$

For each unit increase in vertex count the number of refactorings required increases by 0.0098. The small P value (2.75e-09) for the slop indicates that there is a significant linear relationship between the program size and number of refactorings required to remove top 25% of antipattern instances. In general, this indicates that as the size and complexity of a program increases, it becomes harder to refactor those programs.

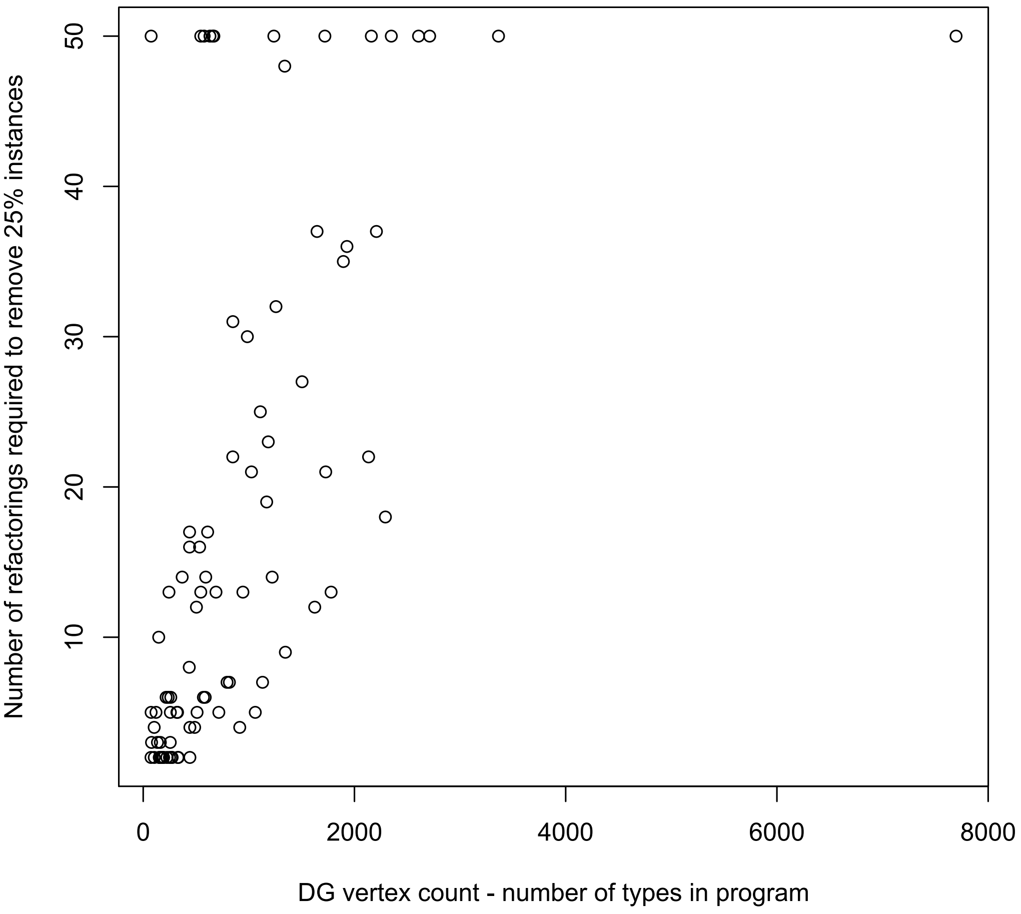


Figure 4.11: Impact of Program Size on Number of Refactorings

4.6.6 Package Merging

Merging all packages into a single package would resolve the problem of circular dependencies between packages. However, this is a trivial solution. On the other hand, merging some of the small-sized packages may reduce coupling. In order to ensure that our move refactorings do not over-merge packages, we counted the total number of packages in programs before and after performing refactorings. We refer to this as package merge ratio. This data is collected for the maximum 50 move refactorings performed across all programs in the dataset.

In 88 programs only 25 programs had packages merged. For 22 programs the ratio of merged packages to total packages ranged between 0.81% and 10%. This small

Program	Total Packages	Merged	Ratio	Classes in Merged Packages
jag-6.1	16	2	12.50	1, 2
sunflow-0.07.2	22	3	13.64	3, 4, 4
freecs-1.3.20100406	12	4	33.34	1, 2, 3, 9

Table 4.6: Result for Merged Packages

proportion of merged packages indicates that our proposed methodology does not over-merge packages and does not compute a trivial solution.

However, there were 3 programs where the ratio of merged packages to total packages was more than 10%. Examples of such ratios include `jag-6.1` (12.5%), `sunflow-0.07.2` (13.64%) and `freecs-1.3.20100406` (33.3%). Table 4.6 shows the three programs where the total to merged packages ratio was more than 10%. We can see from the table that all the packages merged had relatively small number of classes.

4.6.7 Distribution of Move Refactorings

In 88 programs from the dataset, we identified classes that were involved in SCD instances. We performed refactorings in two ways: either move the source class to the target class' package or vice versa. Our selection algorithm chose the best available move refactoring based on the selection criteria. In 49% of the cases the source class was moved to the target class' package, while in 51% cases the target class was moved to the source classes' package.

4.6.8 Refactorability

We define refactorability as the percentage of successfully completed refactoring to attempted refactorings. Given a high-scored edge, there could be four possible outcomes:

1. Both candidates pass graph level pre and postconditions and suppose candidate1 is better than candidate2, so candidate1 is attempted and and it passes the final code level postconditions. Therefore candidate2 is not attempted on the code level.
2. Candidate1 fails graph level pre or postconditions and candidate2 passes all conditions.

Edge	Move Candidate	Result	Refactorability
e1	c1	passed	40%
	c2	not attempted	
e2	c3	failed	
	c4	passed	
e3	c5	failed	
	c6	failed	

Table 4.7: Refactorability Example

3. Candidate2 fails graph level pre or postconditions and candidate1 passes all conditions.

4. Both candidates fail graph level pre or postconditions.

Consider table 4.7 as an example of a program, which has three critical edges. The result column shows that out of 6 possible candidates one is not attempted on the code level. Therefore, the refactorability of this program would be $\frac{passed*100}{attempted}$. In this example the refactorability is 40%.

Figure 4.12 shows the histogram of refacatorability of programs. This figure shows that approximately 50% programs have refactorability range of 10 - 30%. In the figure, there is one program with 100% refactorability. This program required only one refactoring to remove all instances and one candidate of the critical edge passed all conditions and the other candidate was not attempted on the code level.

4.6.9 Success Estimation of Model to Code Refactorings

In the move refactoring experiment, at first we compute a refactoring on the model level and if it passes preconditions, we perform it on the code level. Some of the refactorings that are successful on the model level may not be successfully applied on the code level. In order to check the success rate of model to code refactorings, for every successful refactoring on the model level, we checked whether the refactoring was successful on the code level or not. The results show that, across 88 programs only 7.7 % refactorings failed on source code level postconditions. This means that in approximately 93% cases the predicted refactorings were successfully applied on the code level. The total number of refactoring performed on all programs was 3384 with the average of 33 refactorings per program. The number of failed refactorings was 262. In section 4.6.11, we discuss reasons due to which some refactorings failed on the code level.

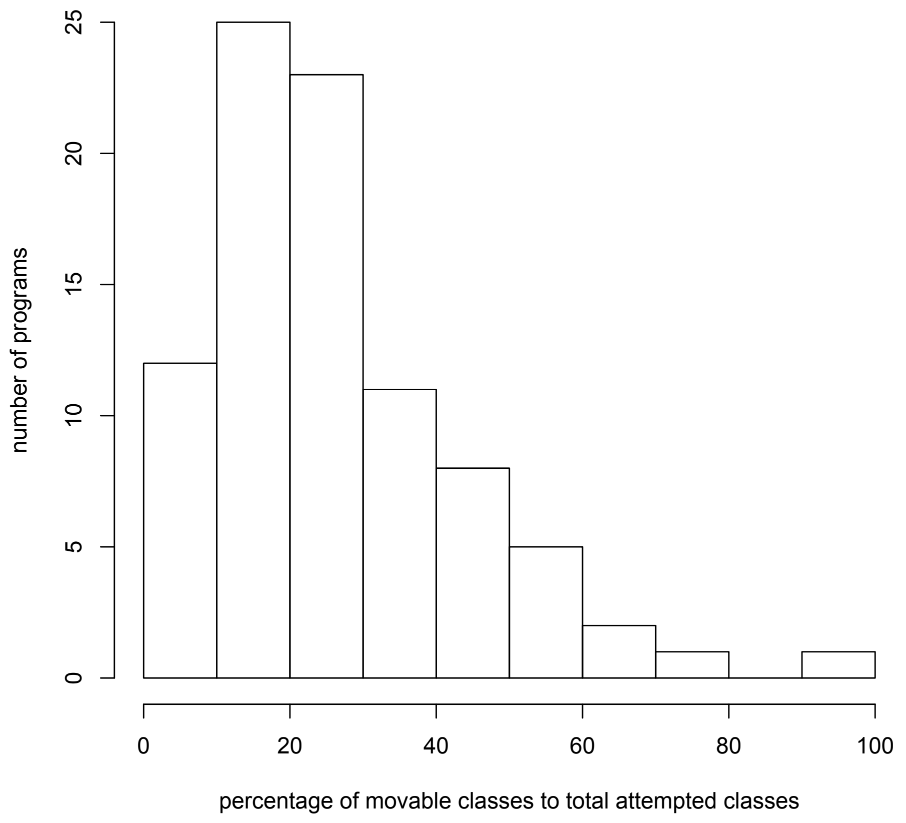


Figure 4.12: Refactorability

4.6.10 Strongly Connected Components Metrics

This section shows how move refactorings have affected strongly connected components across all programs in the dataset. We have computed strongly connected components from the dependency graph of packages in programs. For each program, we have computed SCC metrics before and after performing the top 50 refactorings. Figure 4.13 shows the before and after values of the largest SCC (MAX_SCC) in each program. On average across all programs the before value is 25.33 and the after value is 22.36. This decrease indicates improvement in the SCC structure.

Figure 4.14 shows the average density metric of all refactored programs. Values closer to 1 represent highly dense SCCs. In many programs we can see the drop in the density of SCCs. However, there are several programs where the density has slightly increased. The reason for the increase in density is that when a component is broken into two sub-components, the density of individual components may increase. This does not necessarily mean that the quality of the program decreased. On the contrary, these dense components may represent cohesive clusters of packages that could be merged.

For all refactored programs, the average density value before refactoring is 0.08 with standard deviation 0.116 and after refactoring it is 0.04 with standard deviation 0.046. After refactoring the average density values are more consistent across all programs. We chose arithmetic mean (average) over geometric mean because the latter is appropriate for describing proportional growth, such as, return on investment over a period of time (Anson et al., 2010). It should also be used when data are interrelated. Whereas the arithmetic mean should be used when the data points are not interrelated, i.e., they do not depend on each other (Matuszak, 2010). The programs in the dataset are independent of each other. Therefore, we have used the arithmetic mean.

Figure 4.15 shows the average tangledness of SCCs in programs. A tightly tangled program has a value closer to 1. For many programs the tangledness of programs decreased. For all refactored programs, the average tangledness value before refactoring is 0.16 with standard deviation 0.195 and after refactoring it is 0.08 with standard deviation 0.079. This means that the average tangledness values after refactoring are more consistent across the dataset.

4.6.11 Limitations of the Experiment

There are several limitations in the Eclipse refactoring engine. These limitations are discussed below. If a class cannot be moved due to one of these limitations, we manually put its name in the blacklisted classes (Section 4.4.2.2). We have done this on trial-and-error basis as this does not happen very often.

Constant Folding

The dependency graph is built from the bytecode of programs. In the bytecode, references to static, final primitives and strings are inlined by the compiler after a process called constant folding (Muchnick, 1997). Due to this reason we cannot compute the incoming references to these fields. If such a field is declared protected or private, moving the class to another package would require the field to change the accessibility to public so that it can be accessed by classes from the original package. Consider the following example:

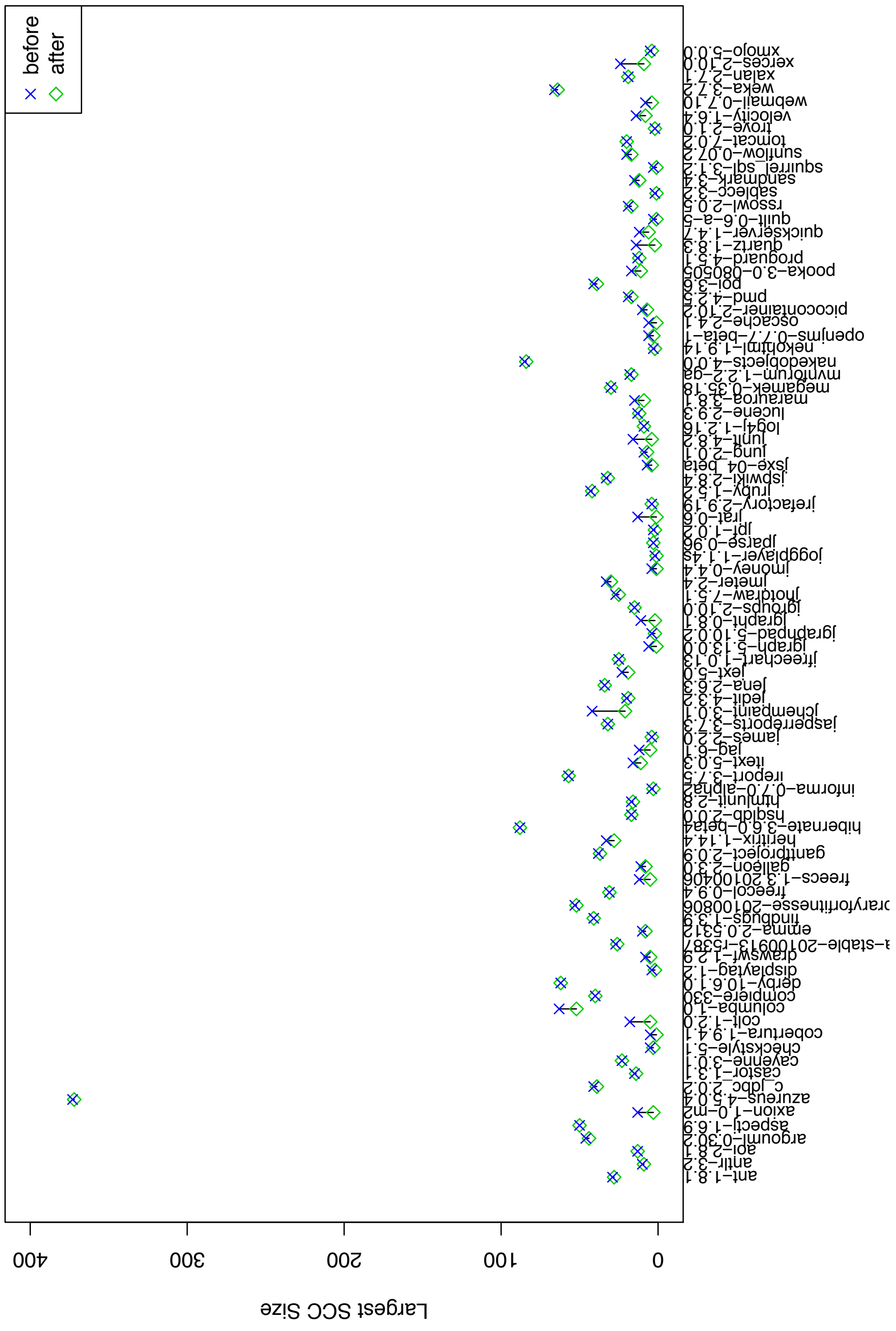


Figure 4.13: Largest SCC in Programs

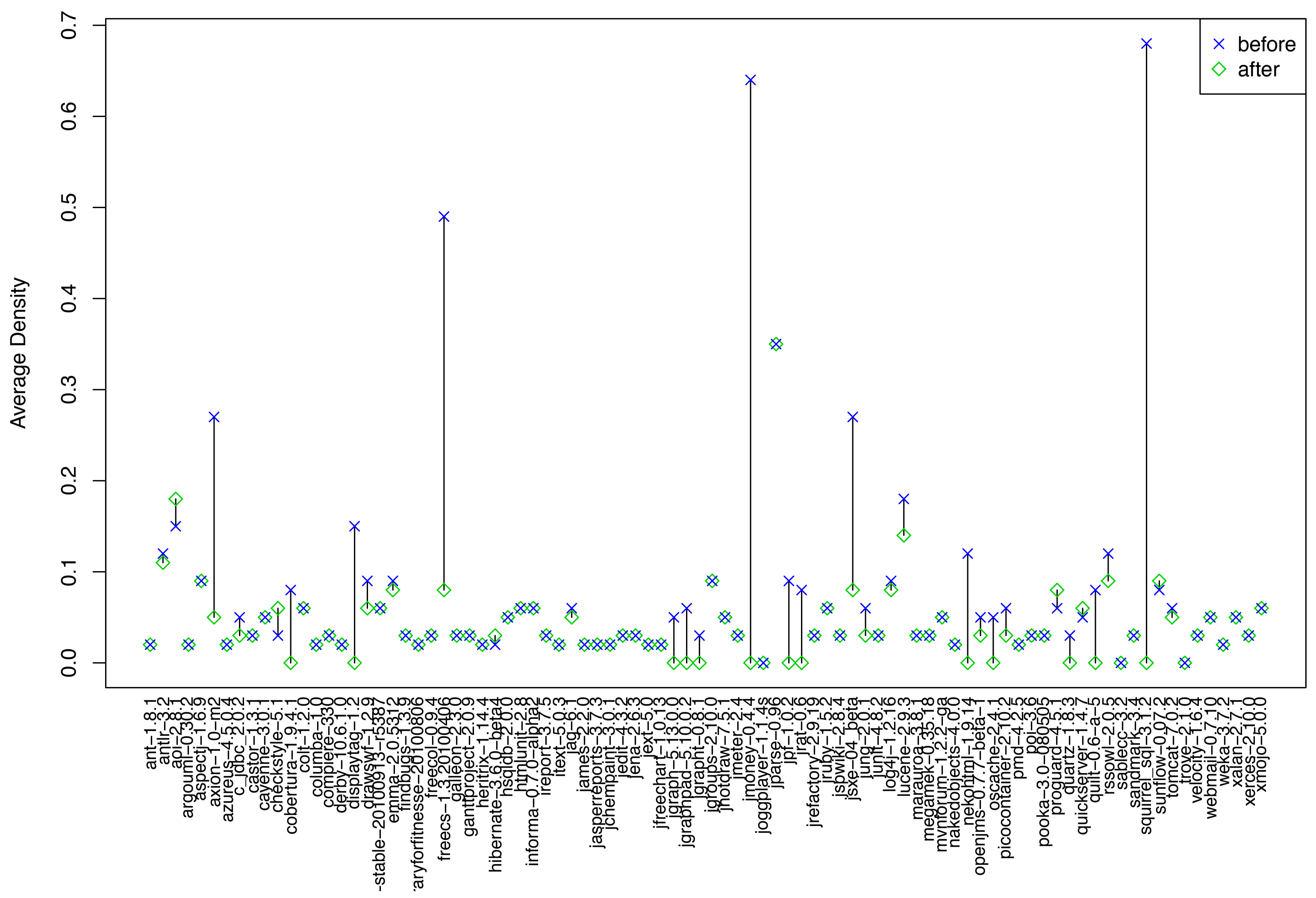


Figure 4.14: Average Density (D) of Programs Before and After Refactoring

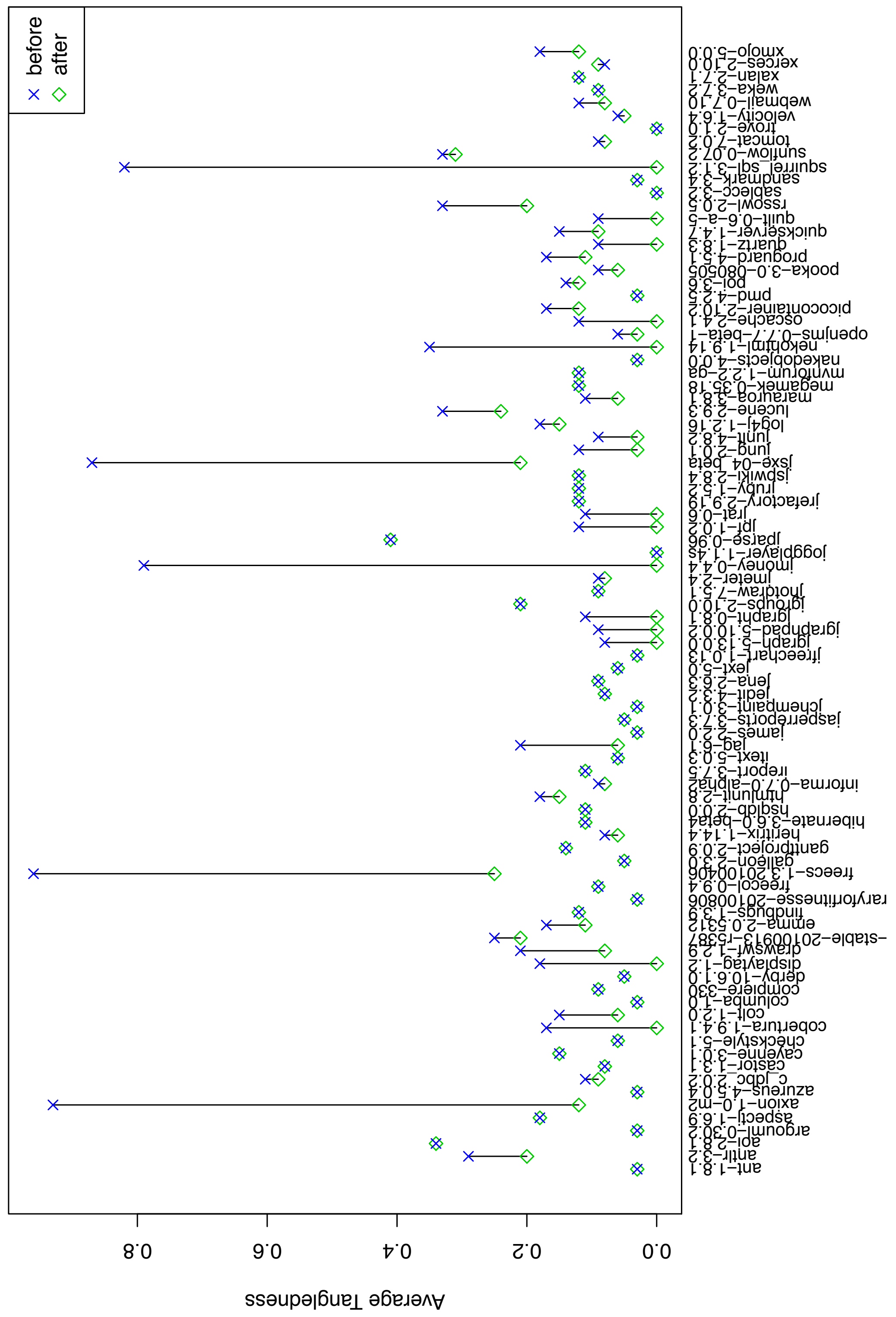


Figure 4.15: Average Tangledness (T) of Programs Before and After Refactoring

```java
package a;
public class A { protected static final MAX_NUM = 100; }
package a;
public class A1 {
   void m() {
      int n = A.MAX_NUM; ... }
}
```

Moving the class A from package a to b would require an additional refactoring i.e. change the accessibility of the field MAX_NUM to public. Since we rely on the Move refactoring provided Eclipse IDE, the compiler generates an error after performing the refactoring.

Name Resolution

In preconditions, before moving a class to another package we check whether a class with the same name exists in the target package. If this is the case, we do not move the class. However, the name resolution error is generated when a class in the target package uses a class with the same name, which is to be moved in the target package. The following example explains the situation:

```java
package a;
public class X {
   ...
   Object o = list.iterator.next();
   ...
}
```

If we move the class `X` to package `b`, which has a class named `b.Object`, then the standard move refactoring considers `java.lang.Object` as `b.Object` and generates a compilation error. This is a limitation in the Eclipse move refactoring.

Invalid Imports

This is an error in the standard Eclipse move refactoring where moving a class removes import reference to an inner class in the moved class. This error is reported on Eclipse bugs website[6]. The error scenario is given below:

[6]`https://bugs.eclipse.org/bugs/show_bug.cgi?id=350205`

```java
package a;
import b.Outer.Inner;
public class ToMove { Inner i; }
package b;
public class Outer {
   public static class Inner {}
}
```

Now, moving the class `a.ToMove` to the b package results in the missing import statement for the Inner class. This generates a compile time error. If we rollback this refactoring, i.e., move the class `ToMove` back to the original package `a`, we also get an invalid import statement which only imports `b.Outer` rather than `b.Outer.Inner`. Therefore a situation like this results in compile time error even after performing the rollback. In order to avoid this situation, we recorded a list of such cases and put those class names in a separate file for their respective projects. While checking the preconditions for the move refactoring these blacklisted classes are skipped, if they are the candidate for move.

4.6.12 Scalability

We have distributed our dataset across 5 computers to expedite the completion time. All computers had Java Runtime 6 installed. Two computer have Intel Core i5 and 4GB RAM, and three computers with Intel Core i7 and 4GB RAM. The average execution time for all 88 programs is 239 minutes. In order to check the impact of program size, we have divided the dataset into two halves based on program size measured by vertex count. The small set has vertex count range of 74 - 570. For this set the average execution time was 16 minutes per program. The large dataset, which ranges from 577 - 7696 vertices, the average execution time of 462 minutes. The big difference of the execution time is due to excessive re-computation of antipattern instances, as this is part of postconditions. Table 4.8 shows 5 programs which took the most of time. For these 5 programs the average execution time was 2531 minutes or 42 hours.

4.6.13 Test Results

Test cases are an important part of a program. They reflect semantics of programs from a developer's point of view. Testing plays a vital role in the refactoring process (Fowler, 1999). Successful tests can be used as postconditions after each refactoring. If a refactoring introduces an abnormal behaviour in the program, testing should

Program	Vertex Count	Edge Count	Execution Time (mins)
azureus-4.5.0.4	7696	44057	4063
compiere-330	2712	16369	3373
aspectj-1.6.9	2607	25019	2147
derby-10.6.1.0	1895	15416	1912
jruby-1.5.2	2134	16375	1161

Table 4.8: Top 5 Programs with Highest Execution Time

uncover that problem. In our framework, test cases were not added to create standard postconditions because not many programs used them[7]. Similarly, different testing frameworks were used, such as JUnit 3, JUnit 4, mock objects, custom tests using the main method of a class. This made it difficult to integrate this concept as a generic postcondition into our framework. However, in order to verify our approach, we have manually executed test cases of 5 programs before and after performing the move refactorings. In the refactoring process we have also refactored test cases. That is, if a class was moved from one package to another, the import declarations of test cases were also updated (where required).

Not all programs in the corpus have test cases. Therefore we manually selected those programs which have high test coverage. Test coverage or code coverage reflects the extent to which a program's source code has been tested (Miller and Maloney, 1963). We used EclEmma[8] tool to compute the statement level test coverage. We expected that test cases with high test coverage are likely to uncover more problems as compared to programs with low test coverage. The successful execution of test cases does not guarantee that the program has no semantic errors. However, it is an indication that what a developer thought to be true is still true after refactoring.

The result of tests execution on 5 programs are shown in table 4.9. In some programs tests generated errors and failures when executed on the original program before refactoring. We did not try to fix those erroneous test cases because a test case which fails before the refactoring should also fail after the refactoring. The results of these 5 programs show that there were no semantic changes caused by move refactorings except in pmd-4.2.5 where the errors after refactorings increased from 31 to 40. In this case the class `net.sourceforge.pmd.MockRule` was moved to the package `net.sourceforge-.pmd.rules`. The `MockRule` class was referred by nine test cases as a string literal. As the standard move refactoring cannot update string literals, therefore these test cases could not be updated and we got "class not found" error. These errors were fixed by

[7]Test cases were available for only 28 programs in the dataset
[8]`http://www.eclemma.org/`

Program	Test Coverage	Errors - Before	Errors - After	Failed - Before	Failed - After	Total Tests
lucene-2.9.3	82.7	19	19	22	22	1292
jgrapht-0.8.1	73.7	0	0	0	0	371
jfreechart-1.0.13	72.6	0	0	0	0	6375
nekohtml-1.9.14	53.5	0	0	5	5	13
pmd-4.2.5	48.9	31	31	52	52	884

Table 4.9: Test Results of 5 Programs Before and After Refactorings

manually updating the class names[9]. After this manual change, the test results were the same as before.

4.7 Summary

In this chapter, we have presented an algorithm that identifies potential refactoring opportunities and executes them on the code level to remove cycles or tangles between packages. We have used the move class refactoring to break cycles. We have also defined pre and postconditions for the move refactoring. This set of pre and postconditions can be further enhanced. For example, the accessibility precondition may be weakened by allowing types and members to change the accessibility to public.

We have evaluated our approach on a set of 92 programs to detect and perform move refactorings in order to break circular dependencies. The results show that our algorithm detects high-impact refactorings that considerably improve the structure of programs. Programs are not only improved in terms of antipattern count metric, but also in terms of other metrics such as those related to strongly connected components. In addition, test results show that refactorings performed by our tool do not change the behaviour of programs.

In order to automatically execute the identified refactorings on the source code level, we have developed an Eclipse plugin - CARE. In CARE, we can safely execute the move class refactoring(s) on a set of programs with a push of a button. In the next chapter, we experiment with alternative strategies to break critical dependencies.

[9]The test results were recomputed after the manual refactoring

Chapter 5

Applying Composite Refactorings

In this chapter, we use composite refactorings to break critical dependencies between classes. We use the count of four antipattern instances as the major fitness function for the analysis of programs. We present an algorithm and a proof-of-concept implementation of the refactoring engine. We also discuss pre and postconditions required to perform refactorings. Our approach uses several refactorings: move class, type generalisation, introduce service locator, and static members inlining and is validated on the Qualitas Corpus set of Java programs.

5.1 Overview

In the previous chapter, we have used package level refactoring to remove critical dependencies between classes. In this chapter, we extend our refactoring framework to include class level refactorings to remove or reorganise dependencies. This means we adopt a composite refactoring strategy, which includes both class and package level refactorings. We also discuss how these refactorings are applied during the refactoring process. We use four class and package level refactorings as discussed in section 3.4. These refactorings are move class, type generalisation, introduce service locators, and static members inlining.

For each program in the dataset, our refactoring algorithm identifies critical dependencies between classes and breaks those dependencies by using the aforementioned refactorings. The critical dependencies are identified by a scoring function, which ranks edges of the dependency graph according to their participation in instances of four antipatterns. As opposed to the previous chapter where instances of only one antipattern were removed, we try to remove instances of four antipatterns (discussed in Section 2.3). These antipatterns are strong circular dependency between packages (SCD), subtype knowledge, (STK), abstraction without decoupling (AWD), and degenerated inheritance (DEGINH).

We do not expect that removing or reorganising dependencies will always succeed. Therefore, refactorings must be safeguarded with a set of pre and postconditions. Our algorithm picks a high-scored edge and checks preconditions for each of the two types involved in the dependency, i.e., the source and the target class. If all preconditions pass, the respective refactoring is tentatively performed on the abstract syntax tree (AST) of the program. Next, postconditions are checked to verify that the refactoring was performed successfully. If postconditions fail, the refactoring is rolled back and the next high-scored edge is picked. Conversely, if postconditions pass, the refactoring is executed on the program's source code and the analysis is repeated until all instances are removed or a certain number of refactorings are performed.

The refactoring algorithm is implemented as an Eclipse plugin, which analyses the source code of programs before applying any refactoring. For this purpose, we build the AST of classes where refactorings are required. We have used the Eclipse Java development tool (JDT) to construct ASTs. In order to perform refactorings on the source code, we have used the Eclipse refactoring toolkit (LTK) API. This API provides the facility of rolling back a refactoring (UNDO Manager) in those cases where the refactoring is not applied successfully. Our approach is validated against the Qualitas Corpus set of programs.

5.2 Background

5.2.1 Type Generalisation

In the literature, several tools and techniques exist that aid developers to use more abstract types. For example, Mayer et al. (2007) have developed a refactoring tool that detects code smells and executes refactorings to remove those smells. In this tool, the supertype hierarchy is displayed to the user in the form of a lattice. The user can choose a relevant refactoring among multiple refactoring suggestions. This process involves human interaction and cannot be fully automated.

Streckenbach and Snelting (2004) have used the KABA refactoring tool, which proposes split classes and move members refactorings. The drawback of this tool is that it cannot modify a program's source code. However, it can be used to modify the bytecode of the program. This makes it difficult to integrate this tool into the software development life cycle. Bach et al. (2007) have developed an Eclipse plugin, which finds better fitting types in programs. This plugin looks for all variable declarations, field declarations, method parameter types and method return types to compute valid supertypes. Once valid supertypes are computed, this plugin generates warnings in the Problem View of Eclipse. Quick fixes are associated with each variable declaration to automatically re-declare a variable with an abstract type. The selection of an abstract type for re-declaring a variable must be done manually. This plugin is not scalable for large programs. For example, the authors reported that for JHotDraw-6.0b1 with 7788 declaration elements, it took 42 minutes to compute generalised types on a 2GHz IBM thinkpad.

There are several approaches based on a metrics suite to determine the refactoring opportunities for types generalisation. Mayer (2003) has analysed the use of interfaces in large object-oriented programs. This study reveals that interfaces are not very popular among programmers. The author defined a metric suite to identify source code places where the use of interfaces should be applied. For this purpose, the author developed an Eclipse plugin, which assists programmers to make use of interfaces for variable and field declarations rather than using concrete classes. In our study, we try to generalise to any compatible supertype and we do not restrict ourself to interfaces. In addition, we perform other refactorings such as introduction of service locators and static methods inlining to break dependencies.

Gossner et al. (2004) has investigated the use of interfaces in the Java Development Kit. They advocate that there is a big opportunity for replacing existing types with their supertypes. The authors have validated the results with the help of several metrics that are related to the use of interfaces within the project. This study doesn't consider abstract classes but only focuses on interface utilisation. In a similar way, Steimann et al. (2003) have presented a study showing that in several large Java projects one out of four variables was declared through its interface. The authors have defined a metric suite related to interface utilisation in object orient programs and proposed refactorings for a better utilisation of interfaces in programs. The metrics definition is very vague and no implementation exists to validate the effectiveness of the approach.

Opdyke and Johnson (1993) have proposed refactorings for creating abstract super classes from concrete classes. The authors identified refactoring steps and provided techniques on how to automate these steps. Tip et al. (2003) have proposed a constraint satisfaction mechanism to verify preconditions for type generalisation refactorings such as *extract interface*, *use super type where possible* and *generalize declared types*. These refactorings have been implemented in the standard distribution of Eclipse. Steimann et al. (2006) have followed a similar approach by developing a tool that can automatically infer types from concrete classes. By selecting a variable the infer type refactoring allows programmers to generate a new minimal interface that can be used to declare that variable. A disadvantage of their approach is that the excessive use of this approach may result in many similar interfaces.

5.2.2 Service Locators

Martin has defined the Dependency Inversion Principle (DIP) (Martin, 1994), which states that "high level modules should not depend upon low level modules. Both should depend upon abstractions". This means a class should depend on an abstract type rather than on a concrete class. However, these abstract types still have to be instantiated using concrete classes. There are several ways to instantiate a concrete class and pass it to the client class exhibiting DIP. One possible solution to locate and utilise implementation classes is a J2EE pattern called service locator[1]. Fowler (2004) has discussed the benefits of using service locators to avoid instantiation problems. He suggests to create a service locator, which has knowledge about a service and its implementations and use the service locator along with a registry to locate and instantiate implementation types. In our previous work, we have used the service locator pattern to achieve modularity in object-oriented programs (Shah et al., 2012).

5.2.3 Static Members Inlining

The inlining technique is primarily used by the Java compiler for the optimisation of program execution, it is also known as constant folding (Muchnick, 1997). This type of inlining is limited to static and final values and string constants, where calls to constants are replaced with their values by the compiler. Inlining is also used as a refactoring technique where fields, methods and classes are inlined into a client class for the purpose of optimisation or for removing a dependency. For example, in order to remove a dependency, Feathers (2004) has suggested a refactoring called *extract and override call* in which the method signature of the invoked method (causing a dependency relation) is copied into the client class and a subclass of the client implements and returns a mock value for testing.

Tsantalis and Chatzigeorgiou (2009) have investigated classes to identify move method refactorings. According to the authors, moving fields and methods within classes may play an important role in achieving low coupling and high cohesion. They propose a novel method for the identification of Feature Envy problems in programs. The Feature

[1]http://www.oracle.com/technetwork/java/servicelocator-137181.html

Envy problem occurs in programs where a method frequently accesses features of a class other than the one in which it exits (Fowler, 1999). The authors introduce a concept of distance between software artefacts, such as fields, methods and classes, which identifies refactoring opportunities. In order to determine the impact of such refactorings an Entity Placement metric is used to identify whether classes are placed correctly in a system. The approach suggested is not fully automated and requires the judgement of system developers to apply relevant refactorings.

Object inlining is a similar technique, where a dependency to an object is removed by inlining its members (Dolby, 1997; Dolby and Chien, 2000). By using this technique calls to `new` object creations (constructor invocations) can be minimised. This minimisation of `new` creations may improve the performance of a system (Ben Asher et al., 2012). For example, we can inline some of the members (that a class X depends on) of a class Y to class X. This would break the dependency from class X to Y. The indirect access to fields and methods in class X would also be replaced by the direct access. However, it is likely that we may create redundancy in the code if a class Z also depended on the same members of class Y which were inlined to class X. Object inlining may significantly change the meaning of classes. Similarly, when members of two classes are merged into a single class, it may also negatively affect the understandability of source code. Due to these reasons we restrict to inlining static members, where we move the required static members of the target class to the source class and leave a delegate method in the target class.

5.3 Algorithm

An overview of the refactoring algorithm is given below:

1. Build the dependency graph from the bytecode of a program.
2. Use the Guery engine (ver 1.3.5) to compute the set of SCD, STK, AWD and DEGINH instances (Dietrich et al., 2012).
3. Compute a list of edges (class dependencies) sorted by score ranking based on their participation in all types of antipattern instances.
4. Parse the program's source code into ASTs.
5. Check preconditions to determine whether a dependency can be removed or not.
6. If the preconditions are satisfied, apply the refactoring on the program's ASTs. Otherwise try the next high-scored edge.

7. Evaluate postconditions to check whether the applied refactoring has introduced any errors.

8. If postconditions are satisfied, update the program's source code. Otherwise, rollback the ASTs to their previous states.

9. Repeat the process until all antipatterns instances are removed or a certain number of iterations are performed (*MAX* is 50 in our case).

The detailed refactoring process is shown in figure 5.1.

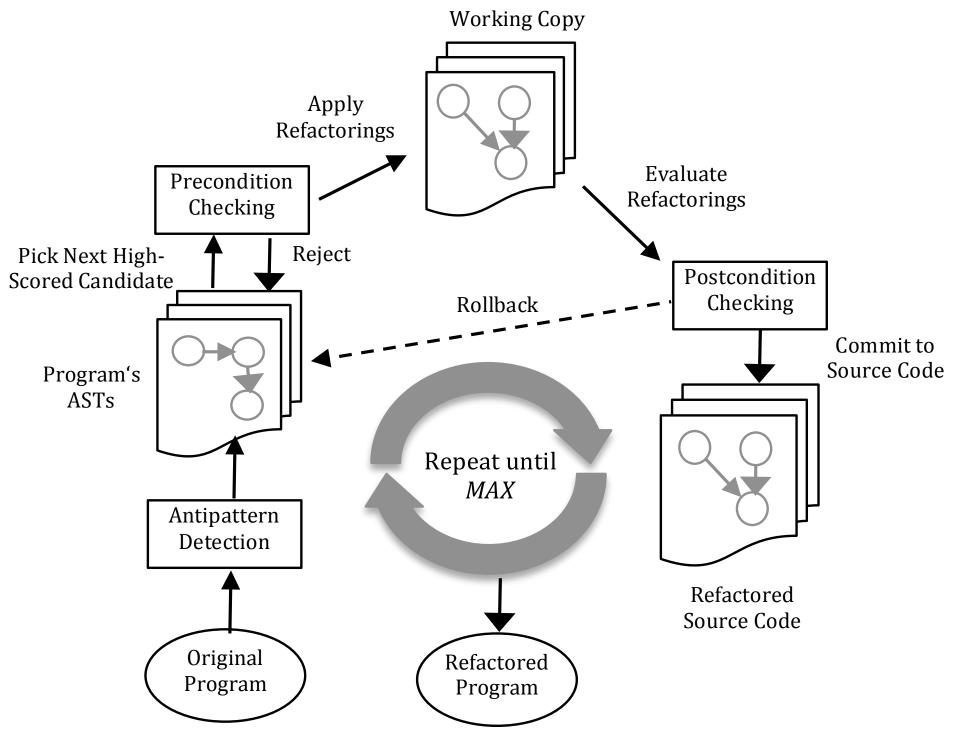


Figure 5.1: Automated Refactoring Process

5.3.1 The Dependency Graph

The first step of the algorithm starts with building the dependency graph of a program. The process of building the dependency graph is explained in section 2.1.

5.3.2 Computing Antipattern Instances

In the second step, the algorithm computes instances of all four types of anitpatterns. These antipatterns are strong circular dependency between packages (SCD), subtype knowledge, (STK), abstraction without decoupling (AWD), and degenerated inheritance (DEGINH).

5.3.3 Computing Edge Scoring

In this step, the algorithm computes score for edges that participate in antipattern instances. The edge scores are sorted so that the highest scored edge is the first candidate for removal. The process of computing edge scoring is discussed in section 2.4.

5.3.4 Parsing Source Code

After the identification of critical edges (class dependencies), we need to analyse the source code to verify whether removing or reorganising these dependencies is possible or not. For this purpose, we extract the Abstract Syntax Tree (AST) of the required classes. We have used the JDeodorant (Tsantalis, 2007) API based on the Eclipse Java Development Tool (JDT) to parse a program's source code. This API provides utility methods that assist in performing static code analysis such as obtaining incoming and outgoing references of a method. The remaining steps of the algorithm are discussed in the implementation section.

5.4 Implementation: CARE - The Eclipse Plugin

We have implemented the above mentioned algorithm as an Eclipse plugin. In this plugin, we have extended the refactoring framework (discussed in section 4.4) to perform composite refactorings.

5.4.1 Implementing Dependency Classification

In order to perform composite refactorings, we have extended the dependency classification mechanism explained in section 4.4.1. The extended class diagram of dependency classification is shown in figure 5.2. As there could be multiple dependency categories in a single class, therefore we have used the composite design pattern for

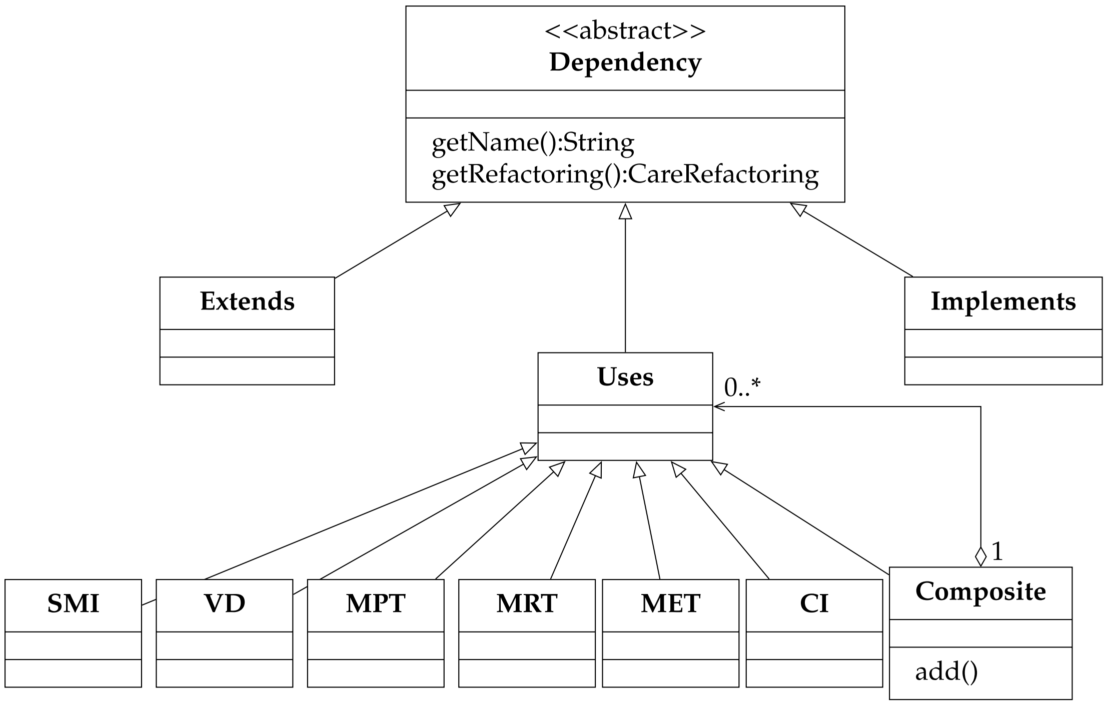


Figure 5.2: Class Diagram of Dependency Categories

the `Uses` class (Gamma et al., 1994). This design pattern allows us to deal with single objects, or a composition of objects, uniformly. When multiple dependency categories exist in the source code, an instance of the `Composite` class is returned.

Given the source and the target class, we check source code of the source class to extract dependency categories. This means we count the occurrences of the target class in the source class and categorise these. In order to extract dependency categories, we have extended the `ASTVistor` class - provided by Eclipse JDT Core. Our `DependencyFinder` class takes the compilation unit of the source class and examines all parts of it to extract different dependency categories.

Table 5.1 shows the default refactoring for each dependency category. This classification can be altered while performing a refactoring. For example, the move refactoring will be attempted when each of the three other refactorings has failed.

Dependency relationships caused by *implements*, *extends*, or *Other* category require change in the logic of a program. Therefore we do not apply class level refactorings, instead we use the move class refactoring to break the respective dependencies.

Dependency Category	**Refactoring**
Variable Declaration (VD) Method Parameter Type (MPT) Method Return Type (MRT) Method Exception Type (MET)	Type generalisation
Constructor Invocation (CI)	Service Locators
Static Member Invocation (SMI)	Static Members Inlining
Extends (EX) Implements (IM) Other	Move Class

Table 5.1: Dependency Categories and their Default Respective Refactorings

5.4.2 Implementing Refactoring Constraints

There are several refactoring specific preconditions, which are described below.

5.4.2.1 Generalize Refactoring Preconditions

These preconditions are checked on the code level (AST of the respective class).

No Valid Supertype: The first precondition is related to type generalisation. The type of a variable, field, method parameter or return type cannot be generalised if the class members invoked on that variable or field are not part of any supertype's interface. Consider the following example where a method parameter type is generalised. In this case, class `A` can invoke the method `size()` on both types: `java.util.Vector` and `java.util.Collection`.

```
//Before refactoring
class A {
   public int foo(java.util.Vector nums) {
      int size = nums.size(); ...
   }
}
//After refactoring
class A {
   public int foo(java.util.Collection nums) {
      int size = nums.size(); ...
   }
}
```

The above refactoring would fail if the class `A` invoked `nums.get(i)` in the method body of `foo()`. In this case, replacing `java.util.Vector` with its supertype `java.util.-Collection` is not possible because the method `get(int i)` is not a part of `java.util.-Collection` interface.

Leaking Reference: The second precondition states that a type cannot be generalised when a reference to the the target type is leaked. For example, in the following code the vector object `nums` is passed to a method of another class. Therefore we cannot determine which part of the interface on the vector object reference is used by the `process` method of the class B. While it is possible to follow this references in principle, the existing *Generalize Declared Type* Eclipse refactoring does not support this.

```java
class A {
   public void foo(java.util.Vector nums) {
      new B().process(nums);
   }
}

class B {
   public void process(java.util.Vector v) {
      ...
   }
}
```

Generalise Non-Public Artefacts: The third precondition states that if a target type is used as the declaration type of a public field or used as a return type of a public method, we cannot generalise that target type. Doing so may break the external client code that is not visible during refactoring. This refactoring is bound to the type of program being refactored. If we are trying to refactor a program which is used (e.g., as a library) in different contexts, this precondition should be enabled. Stand alone applications that are not used programmatically by other programs do not require this precondition. In the context our work, we apply refactorings on a large dataset where it is hard to differentiate between the two types of programs due to lack of meta data in the corpus. Therefore, we ignore this precondition and apply the generalize refactoring on all modifiers.

5.4.2.2 Inline Refactoring Preconditions

Self Instance Creation: This precondition checks the method body of the target method and confirms that there does not exist a new instance creation of the target type. For instance, this is the case when the singleton design pattern is used. The following example explains the situation. In this example, even if we move `staticMethod()` to the class `A`, the dependency on the class `B` would still exist in the class `A`. This precondition is also checked on the code level.

```java
//Before Refactoring
class A {
   ...
   void m() {
      B.staticMethod();
   }
   ...
}
class B {
   ...
   public static void staticMethod() {
      B b = new B();
   }
   ...
}
//After Refactoring. The dependency on B still exists after inlining
class A {
   ...
   void m() {
      B b = new B();
   }
   ...
}
```

Class Boundary: This precondition states that a static member cannot be inlined if it accesses any member of its super class.

5.4.2.3 Move Class Preconditions

A class is not moved if its visibility is restricted to the package level and it is referenced by other classes within the original package. The move refactoring is not performed where the target package contains a class with the same name. If the class to be moved is an inner class, its compilation unit (all types within the outer class) is moved. A detailed discussion on the move class refactoring preconditions is given in section 4.4.2.

5.4.2.4 Postconditions

We have used the same postconditions as described in section 4.4.2.4. The first postcondition states that the program should compile successfully after the refactoring. For example, moving a non-public class to another package results in compilation error, if the class is referenced in the original package. The second postcondition is related to the instance count before and after refactoring. The instance count after refactoring should be less than or equal to the instance count before refactoring. A refactoring is only considered successful when it passes these two postconditions. If either of the postconditions fails, the whole refactoring is rolled back.

5.4.3 Implementing Refactorings

We have extended the refactoring mechanism described in section 4.4.3 to accommodate composite refactorings. This is because we use four refactorings to deal with dependency categories: introduce service locators, type generalisation, static memebers inlining, and move class. The class diagram of refactorings is shown in figure 5.3.

5.4.3.1 Generalize Refactoring

The `Generalize` refactoring is built on top of the standard Eclipse refactoring `Generalize Declared Type`, implemented via the `ChangeTypeRefactoring` class. In the *Generalize Declared Type* refactoring, we have to manually select a declaration type and see whether this can be replaced with one of its supertypes. The `Generalize` refactoring *automatically* replaces all target type declaration elements within a class with a specific compatible supertype.

The `ChangeTypeRefactoring` class computes a list of all possible supertypes with which a declaration type can be replaced. This class supports type generalisation on declaration of fields and variables (VD), method parameters (MPT) and method return types (MRT), but not on method exception types (MET). The MET type of dependency is currently not supported.

In a source class, a dependency to the target type T may occur as VD, MPT and MRT

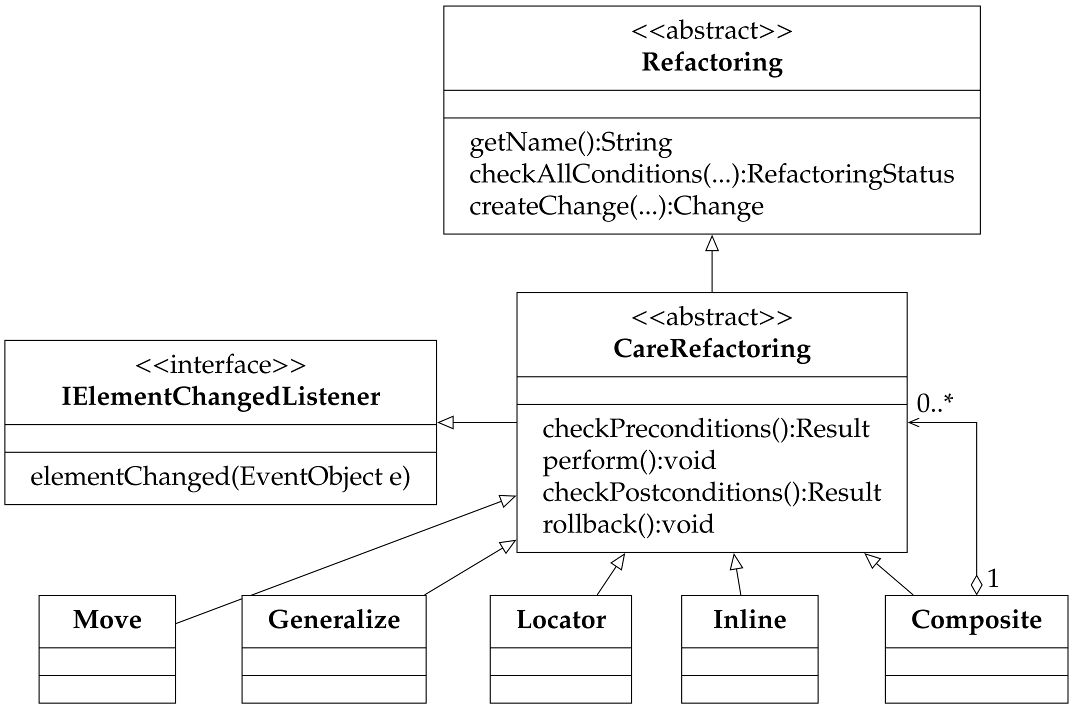


Figure 5.3: Class Diagram of Composite Refactorings

simultaneously. The generalize refactoring must be applied to all such dependency categories in order to remove the dependency to T. If any of the three dependencies (VD, MPT, or MRT) cannot be generalised, the whole refactoring fails. In our approach, we choose a common supertype and replace the target type T with that supertype for each dependency category. The following steps describe the process through which a specific supertype is selected for the target type T.

1. For each VD, MPT, and MRT dependency to T, compute the most general possible type and add it to a set of *PossibleSuperTypes* in no particular order. This is done using the standard Eclipse refactoring class `ChangeTypeRefactoring`.

2. Next, a common supertype is selected from the set *PossibleSuperTypes*. This supertype must be the least general one among the set to avoid API access problems. Due to multiple interface inheritance, a least general supertype does not always exist. In this case, the refactoring is abandoned. For the purpose of choosing a least common type, the original supertype hierarchy is extracted *(AllSuperTypes)* using Eclipse JDT Core API *ITypeHierarchy*. This API returns an array of all supertypes of the current type in bottom-up order. This means `java.lang.Object` would be the last element in the list.

3. Iterate over *AllSuperTypes* from the first *type* to the last. If the *type* exists in the

set *PossibleSuperTypes*, choose this type as a supertype to replace the old type in all identified places of the source class and stop the iteration. This would select the least general type among the set of *PossibleSuperTypes*.

5.4.3.2 Locator Refactoring

The `Locator` refactoring is built on top of the standard Eclipse refactoring *Introduce Factory*. The `Locator` refactoring replaces all constructor invocations of the target type in the source class with the factory pattern. This refactoring creates a factory method in the factory class, which returns an instance of the target type. In this experiment, `registry.ServiceLocator` class is created for each program. This class serves as a global factory class for the instantiation purpose. Consider the following example:

```java
//Before Locator Refactoring
public class A {
   private B bObject = null;
   public void m() {
      bObject = new BImpl();
      ...
   } ...
}
//After Locator Refactoring
public class A {
   private B bObject = null;
   public void m() {
      bObject = registry.ServiceLocator.createBImpl();
      ...
   } ...
}
```

The implementation of the service locator refactoring can be further improved by using other APIs such as Java service loader or Java reflection. This will remove any explicit dependency from the target class and the service locator class. Therefore, when antipattern instances are recomputed to measure the impact of refactoring, these types are ignored.

5.4.3.3 Inline Refactoring

The `Inline` refactoring is built on top of the standard Eclipse move refactoring. The `Inline` refactoring determines all static members of the target class and moves those members to the source class. Delegate methods are created in the target class for moved static methods. Inlining is recursive, e.g., a static method calls another static method within the same class. In that case both methods will be inlined. However, the `Inline` refactoring does not follow references beyond the class boundary, e.g., methods invoked on supertypes.

5.4.3.4 Move Class Refactoring

The `Move` refactoring is built on top of the standard Eclipse refactoring `MoveRefactoring`. In this refactoring a class is moved from one package to another. First, the algorithm performs move refactoring on the dependency graph to evaluate its impact. Next, it is executed on the code level.

5.4.3.5 Composite Refactoring

A composite refactoring is a combination of individual refactorings. The standard Eclipse refactoring framework doesn't support the composite refactoring. Occasionally, it is essential to use a composite refactoring to perform a task, e.g. to remove a dependency to a type, it may be required to use several individual refactorings such as generalize, locator, and inlining. For example, `ArrayList list = new ArrayList()` can be refactored as `List list = ServiceLocator.createList()`. This refactoring required the combination of type generalisation and service locator.

Since our individual refactorings are built on top of standard eclipse refactorings, each refactoring executes changes on its own working copy of the compilation unit. This makes it difficult to perform composite refactorings. The AST modifications overlap when changes in multiple working copies are committed to the original source code. In order to deal with the synchronization problem of multiple working copies of the ASTs, we attached an interface `IElementChangedListener` to the `CareRefactoring` class. This listener interface receives notification about any changes made to Java elements and informs other refactorings in the queue to synchronize their working copies of the ASTs.

5.5 Experiment

We have evaluated our approach on 92 programs in the dataset. First, we discuss a couple of examples from results and then we explain overall results obtained from 92 programs.

5.5.1 Examples

The first critical dependency identified in Findbugs-1.3.9 is the reference from `edu.umd.cs.findbugs.ShowHelp` to `edu.umd.cs.findbugs.gui.FindBugsFrame`. This dependency is caused because the `ShowHelp` class invokes a static method `showSynopsis()` on the target type `FindBugsFrame`, as shown in listing 5.1. This particular dependency is also a reference from a core package `edu.umd.cs.findbugs` to a presentation package `edu.umd.cs.findbugs.gui` and therefore violates the design principle that logic should not depend upon presentation. This dependency was involved in 113579 SCD instances, 185 STK instances and 1811 AWD instances. Therefore, removing this dependency reduced the total number of instances from 277091 (100%) to 161517 (58%).

```java
//Before Refactoring
public class ShowHelp {
   ...
   public static void main(String[] args) {
      ...
      FindBugsFrame.showSynopsis();
      ...
   }
   ...
}
```

Listing 5.1: Source Code of ShowHelp Class

The refactored version of the old code causing the dependency is shown in listing 5.2. A delegate method is created in the target type `FindBugsFrame`, which invokes the `showSynopsis()` method in the source class.

```java
//After Inline Refactoring
public class ShowHelp {
   ...
   public static void main(String[] args) {
      ...
      showSynopsis();
      ...
   }
   public static void showSynopsis() {
      System.out.println("Usage: findbugs [general options] [gui
         options]");
   }
}
```

Listing 5.2: Source Code of ShowHelp After Refactoring

Another example of a critical dependency identified in JHotDraw is a reference from `org.jhotdraw.draw.TextFigure` to `org.jhotdraw.draw.tool.TextEditingTool`. This dependency was involved in 18 SCD instances and 7 AWD instances. The listing 5.3 shows the variable declaration reference in `TextFigure` that causes this dependency.

This particular dependency is a clear violation of the Dependency Inversion Principle because `TextFigure` is using an abstract type `Tool` (an interface) as well as a concrete type `TextEditingTool` (an implementation class of `Tool`). This dependency is safely removed by abstracting the old variable declaration type to `Tool` and by replacing the constructor invocation call with a call to `ServiceLocator` to obtain the implementation class of the interface `Tool` (see Listing 5.4).

```
//Before composite refactoring (generalize and locator)
public TextFigure ... {
   ...
   public Tool getTool(Point2D.Double p) {
   if (isEditable() && contains(p)) {
         TextEditingTool t = new TextEditingTool( );
         return t;
      }
   return null;
  } ...
}
```

Listing 5.3: Source Code of TextFigure Before Refactoring

```
//After composite refactoring (generalize and locator)
public TextFigure ... { ...
   public Tool getTool(Point2D.Double p) {
   if (isEditable() && contains(p)) {
```

```
            Tool t = ServiceLocator.createTextEditingTool( );
            return t;
        }
    return null;
  } ...
}
```

Listing 5.4: Source Code of TextFigure After Refactoring

5.5.2 Impact of Refactorings on Instance Count Metric

In order to evaluate our approach, we have used the Qualitas Corpus version 20101126r (Tempero et al., 2010). Figure 5.4 shows the decrease in antipattern instances over 92 programs after increasing numbers of refactoring are performed[2]. We have used the same plotting mechanism as described in section 4.8.

It can be noticed from the figure that after 20 refactorings the median value over all observations drops below 70%. This means that for more or less half of the programs, only 20 refactorings are required to remove 30% of instances. Similarly, only 8 refactorings are required to remove the first 20% of instances. The boxplot has several outliers. These outliers include programs where the total number of antipattern instances dropped close to zero with a few initial refactorings, such as, jag-6.1, jsXe-04, freecs-1.3.20100406. In some programs the instance count remained unchanged, for example, jFin-DateMath-R1.0.1, fit java-1.1, and javacc-5.0.

5.5.3 Refactoring Simulation on Model vs Refactoring Application on Code

Figure 5.5 shows the comparison between the refactoring simulation on the graph level and the refactoring application on the code level. The two lines represent the median value. This figure allows us to assess the extent to which high impact refactorings can be automated.

There are 50 iterations and for each iteration, the refactoring on the graph level is performed by removing the highest-scored edge without checking any pre and post-conditions. This is why we get a very steep curve representing the decline in the number of instances. However, not every edge can be removed on the code level, as some pre or postconditions may fail. Consequently, our algorithm iterates over a list of high-scored edges until it reaches one which passes all conditions. Therefore, the

[2]Out of those 92, one program (jasml-0.10) had zero instances of all four antipatterns.

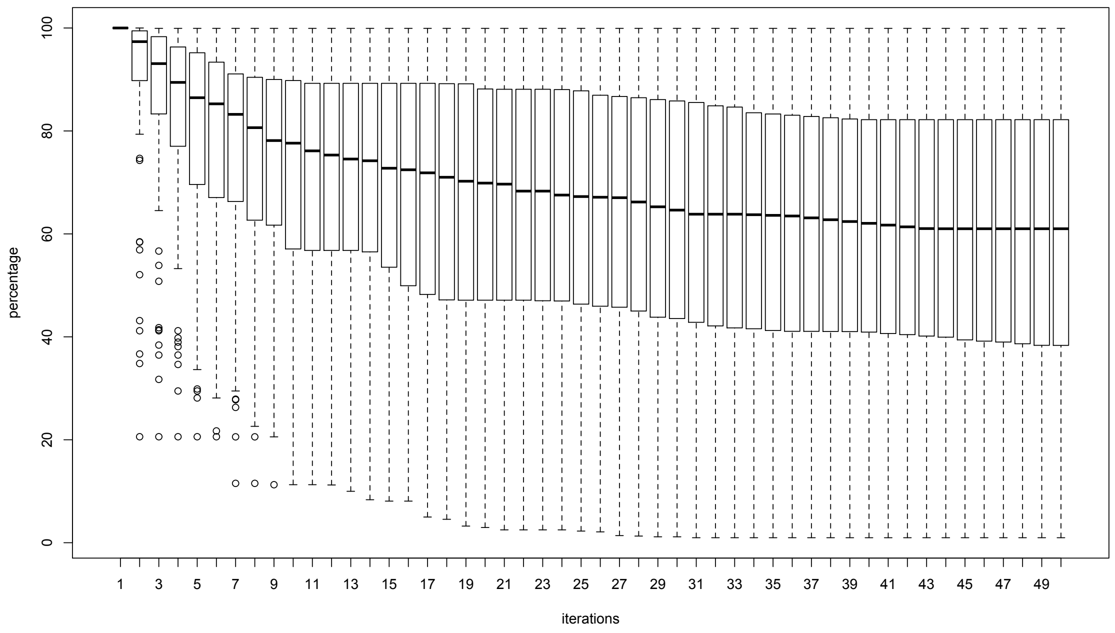


Figure 5.4: Decrease in Instance Count Metric After Refactorings

curve generated by the refactorings on the code level is less steep than graph level simulation. This defines a lower bound of what is possible. The gap between the two lines can be further reduced by weakening the preconditions. For example, we may allow those refactorings that require further refactorings such as change accessibility. In that case the gap between the two lines will be minimised.

5.5.4 Refactoring Types Applied

Figure 5.6 shows different types of refactorings that were successfully applied across all programs. On top is the move refactoring, which is applied in 31% cases. The composite refactorings constitute to 14% of the total refactorings applied.

5.5.5 Strongly Connected Components Metrics

This section shows how the composite refactorings have affected strongly connected components across all programs in the dataset. These metrics are defined in section 4.5. For each program, we have computed SCC metrics before and after performing the top 50 refactorings. Figure 5.7 shows the before and after values of the largest SCC (MAX_SCC) in each program. On average across all programs the before value is 24.07 and the after value is 22.50. This decrease indicates improvement in the SCC structure.

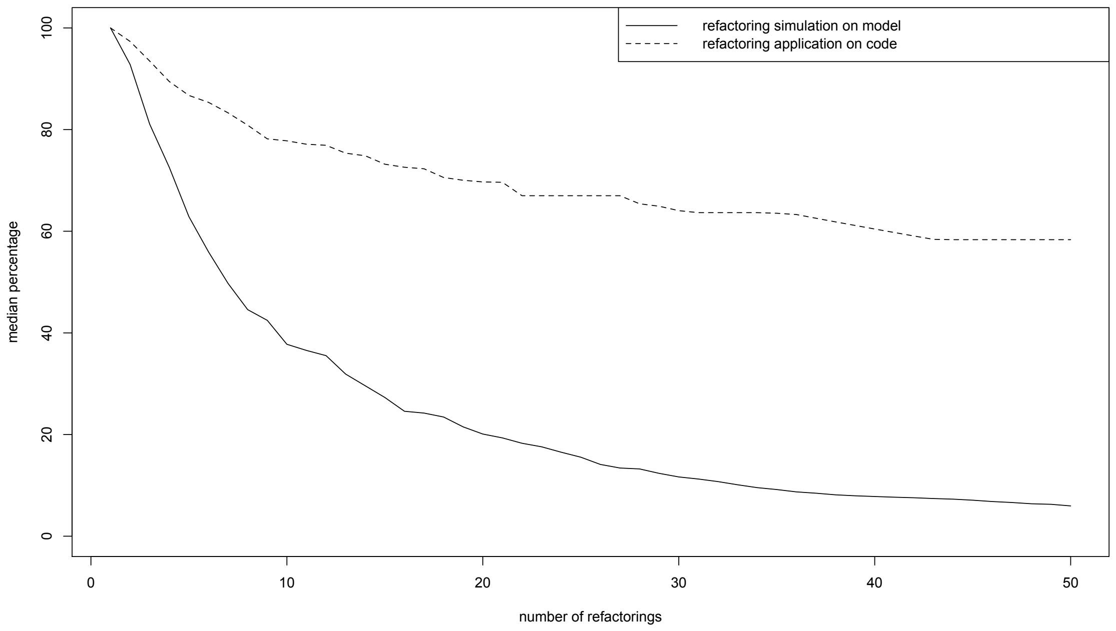


Figure 5.5: Decrease in no. of Instances: Comparison on Model (graph) and Code levels

Figure 5.8 shows the average density metric for all refactored programs in the dataset. Values closer to 1 represent highly dense SCCs. In many programs we can see the drop in the density of SCCs. However, there are several programs where the density has slightly increased, such as marauroa3.8.1 and quickserver1.4.7. This occurs because when a component is divided into sub-components, the density of individual components may go up. This does not necessarily indicate that the quality of the program has decreased. On the contrary, these dense components may represent cohesive clusters of packages that could be merged. For all refactored programs, the average density value before refactoring is 0.07 with standard deviation 0.014 and after refactoring it is 0.04 with standard deviation 0.035. This indicates that after refactoring the average density values are more consistent across all programs.

Figure 5.9 shows the average tangledness of SCCs in programs. A tightly tangled program has a value closer to 1. For many programs the tangledness of programs decreased. For all refactored programs, the average tangledness value before refactoring is 0.16 with standard deviation 0.194 and after refactoring it is 0.08 with standard deviation 0.068. After refactoring average tangledness values are more consistent across the dataset.

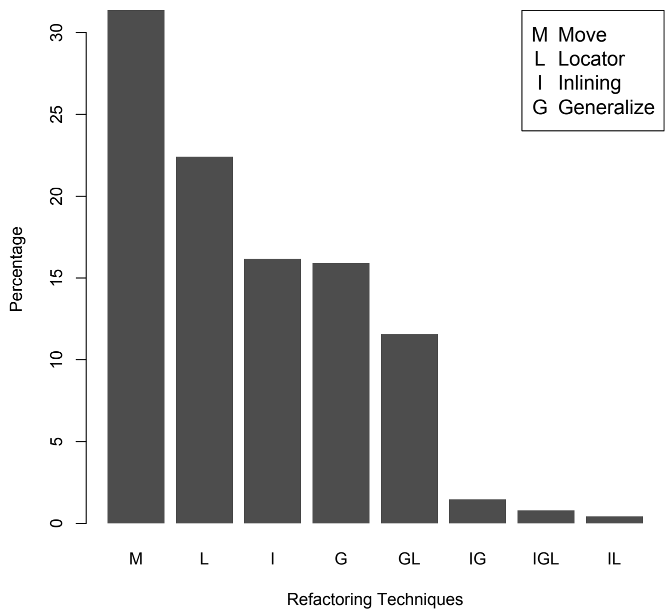


Figure 5.6: Refactoring Types Applied

5.5.6 Test Results

We have manually executed testcases of five programs, as explained in section 4.6.13. Table 5.2 shows the result of tests execution on these programs. The results show that there were no semantic changes caused by composite refactorings.

Program	Test Coverage	Errors - Before	Errors - After	Failed - Before	Failed - After	Total
lucene-2.9.3	82.7	19	19	22	22	1292
jgrapht-0.8.1	73.7	0	0	0	0	371
jfreechart-1.0.13	72.6	0	0	0	0	6375
nekohtml-1.9.14	53.5	0	0	5	5	13
pmd-4.2.5	48.9	31	31	52	52	884

Table 5.2: Test Results of 5 Programs Before and After Refactorings

5.6 Summary

In this chapter, we have presented an algorithm which removes critical dependencies from Java programs. We have developed an Eclipse plugin, which automates the refactoring process. This plugin is able to perform composite refactorings. In general,

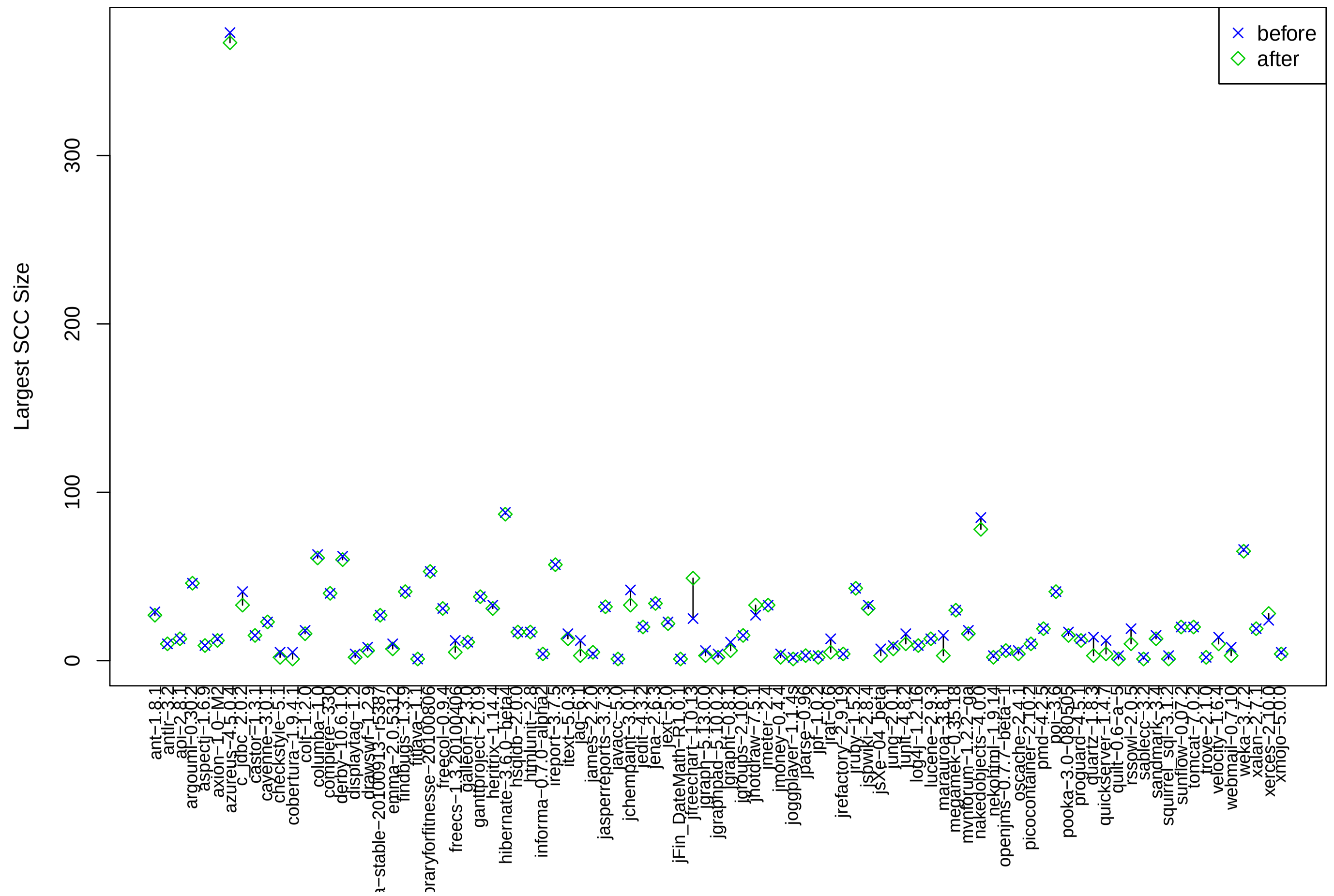


Figure 5.7: Largest SCC in Programs

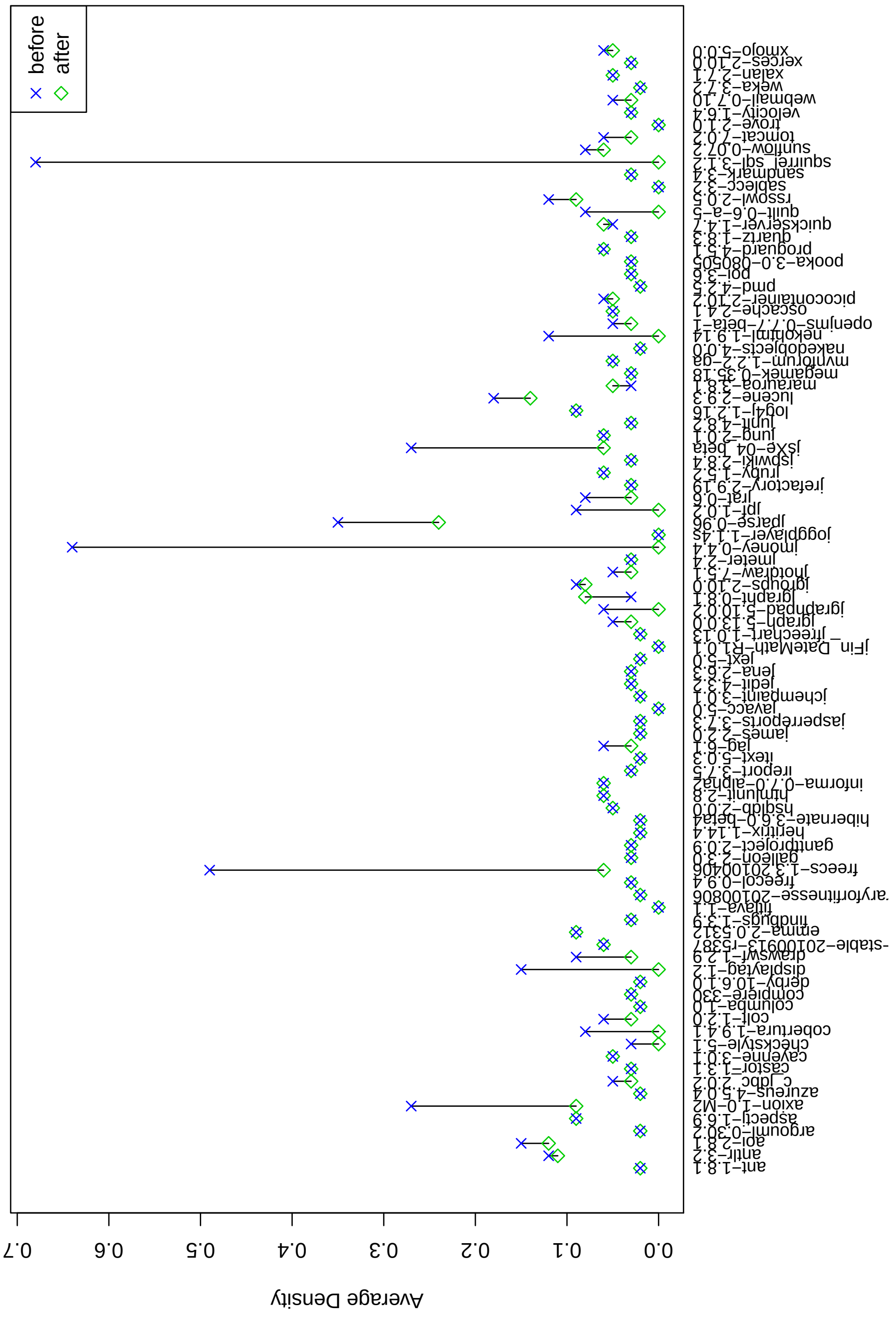


Figure 5.8: Average Density (D) of Programs Before and After Refactoring

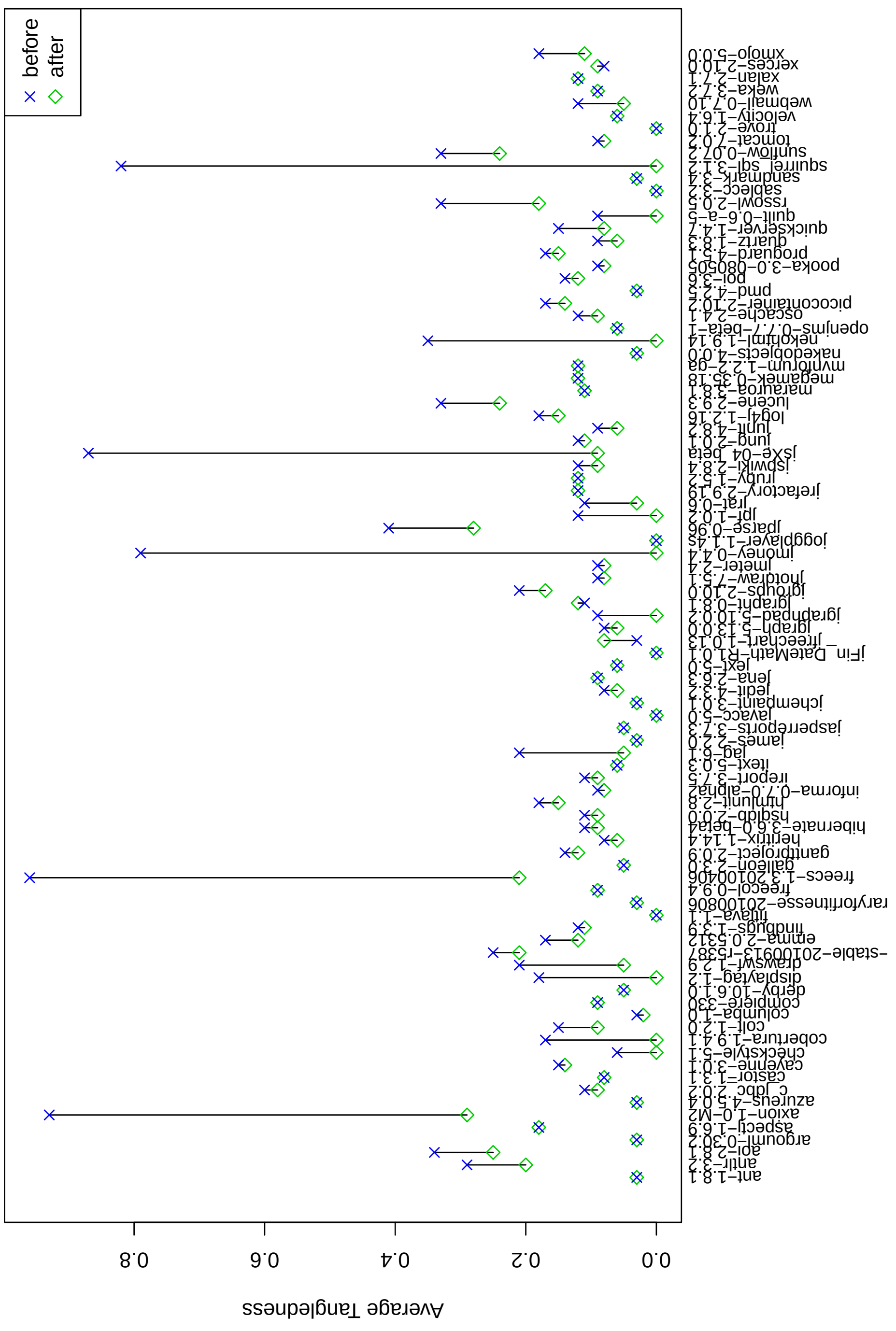


Figure 5.9: Average Tangledness (T) of Programs Before and After Refactoring

it is not possible to remove all types of dependencies automatically. However, there are certain cases where it is possible to safely remove undesired dependencies. All the code level changes performed by the plugin are recorded to be reviewed by a developer of the program. The results indicate that automation of refactoring is possible when all pre and postconditions are satisfied.

The experimental results presented in this chapter are not as good as for Chapter 4. This is because the space of possible application of refactorings is much wider. In this experiment, we tried to remove four types antipatterns by using a set of four refactoring techniques. The results presented can be further improved by weakening preconditions, such as following leaking references, and by strengthening the implementation of the basic code level refactorings such as move refactoring. Further postconditions can be added to improve the safety of refactorings. For example, the successful execution of testcases can be added as a postcondition after each refactoring. Similarly, standard design metrics can be used to check software quality after refactoring. We suggest using this tool in conjunction with a code repository, such as Subversion, in case the developer wants to rollback a particular refactoring after the automated refactoring process is completed.

Chapter 6

Conclusions and Future Work

In this work, we have presented methods, tools and techniques providing a concrete solution for restructuring critical dependencies in object-oriented systems. The major contribution of our work is that it automatically improves the structure of Java programs by identifying and applying dependency-breaking architectural refactorings. These refactorings remove software design problems represented as antipatterns in programs. Our approach not only detects antipatterns, but can resolve them using appropriate refactorings, thus providing a complete refactoring solution. In this chapter we discuss the research questions and the limitations of our work. Here we also provide an overview of the research contributions and directions for future work.

6.1 Research Questions

The thesis has investigated the following three questions:

1. Can model level dependency-breaking refactorings be automatically translated into source code refactorings?

2. How can we define and evaluate constraints on refactorings to preserve the correctness of the program being refactored?

3. To what extent can these dependency-breaking refactorings be automated?

6.1.1 Can model level dependency-breaking refactorings be automatically translated into source code refactorings?

The goal of the first research question was to determine whether the automation of dependency-breaking refactorings is possible or not. To address this question, we investigated different refactoring techniques. There are several techniques of breaking dependencies. We investigated the use of the following: introduction of service locators, type generalisation, static members inlining, and move class. When these techniques can be applied, automation is possible. However, these techniques cannot always be applied. To refactor, the model of a program has to be built from its source code and has to be reasoned about. This may fail in the following ways:

1. the models are so complex that reasoning is computationally too expensive. For example, this occurs when we try to compute the used APIs recursively.
2. the models do not completely and correctly represent the semantics of the program. For instance, this is the case when reflection and other late binding techniques are used used in programs.

We have developed several algorithms for our study. Section 4.3 presents the algorithm for the move class refactoring, where critical dependencies were identified on the model level and then removed on the source code level. We have evaluated our approach on 92 open source programs. The results indicate that the automation is possible in many cases (see quantification in 1.1.3). Similarly, in section 5.3 the algorithm for composite refactorings is presented, where all aforementioned refactoring techniques are used to break dependencies between classes. The proof-of-concept implementation of the algorithm was applied to the same dataset and it proved again that the automation of architectural refactoring is possible to some extent.

6.1.2 How can we define and evaluate constraints on refactorings to preserve the correctness of the program being refactored?

The next question is how to safeguard the investigated refactorings. We have chosen pre and postconditions for this purpose. We have defined refactoring specific pre and postconditions that ensured the safe execution of the refactoring. An important part of refactoring is that it doesn't change the behaviour of the program. Our refactoring tool ensured that not only preconditions are checked, but postconditions are also checked to ensure the behaviour preservation. Refactorings were implemented to run as transactions. This enabled us to safely roll back refactorings when postconditions failed. In section 4.4.2 a set of pre and postconditions related to the move class refactoring are presented. Similarly, preconditions for other refactoring techniques are presented in section 5.4.2.

6.1.3 To what extent can these dependency-breaking refactorings be automated?

We have evaluated our approach on a large set of open source programs to determine the extent to which model to code refactorings can be automated. We have not investigated all dependencies, but only critical dependencies according to our definition. Therefore, this question has been answered with respect to a particular model that is used to select dependencies to be removed. The model we have used is based on the edge-scoring algorithm (see Section 2.4). This allowed us to simulate the best case scenario and to benchmark the refactorings against this. The simulation provided us a lower-bound to be compared with the application of refactorings on the code level.

We have conducted several experiments to determine the difference between simulation on the model level and application on the code level. The comparison between the two scenarios is presented in section 4.6. This experiment was conducted to remove circular dependencies between packages antipattern by using the move class refactoring. The results of this experiment indicate that a significant fraction of dependencies can be removed. In this case, 30% of antipattern instances were removed in 5 refactorings on the model level, while it took 14 refactorings to remove the same number of antipatterns on the code level.

Another experiment was conducted to remove four types of anitpatterns (Section 2.3) by using a combination of aforementioned refactoring techniques. The result of simulation and application of refactorings is presented in section 5.5. To remove top 30% of antipattern instances, it took 5 refactorings on the model level, whereas it took 20 refactorings to remove the same number of instances on the code level. We believe that this gap between actual refactorings and the lower-bound can be further narrowed down by weakening preconditions and by adding new refactoring types for breaking dependencies.

6.2 Threats to Validity

6.2.1 Dataset Selection

We have evaluated our approach on the Qualitus Corpus (ver. 20101126). This dataset has a wide range of real-world programs. We chose this dataset because all programs in this dataset are written in Java programming language and are readily available. Java is one of the most popular object-oriented programming languages. It has been widely used for academic studies. However, there are other software repositories and datasets that have been used for empirical investigations (Pek and Lämmel, 2013). For example, Software-artifact Infrastructure Repository (SIR) contains Java, C, C#, and C++ software system along with testing data (Do et al., 2005). There are some common Java programs in our dataset and SIR, such as ant, derby, jboss, jmeter, and log4j. We do not expect a major difference in results for SIR repository (Java programs), but it would be interesting to test our approach on other repositories and programs in different programming languages.

6.2.2 Correctness of Refactored Programs

There exist several models or constraints against which the program correctness can be verified. These constraints can be checked with the help of different tools. We discuss three such models as follows:

1. Java Type System: The type system of a programming language defines rules, which are used to enforce that a program is written correctly (Pierce, 2002). In order to ensure the correctness of programs, types should be checked for consistency with the help of a tool. In the Java type system, the Java specification language imposes constraints, which are enforced by the Java compiler. This process happens statically, i.e, during compile time.

2. JUnit Testing: Software testing is an activity to evaluate a program with respect to its specifications (Beck, 2002). It is used to verify the correctness of programs and uncover bugs. Test cases represent the expected behaviour of the program from developers perspective. From the verification point of view, test cases can be considered as constraints and a tool is required to check them. For example, in Java programs JUnit test cases can be executed with a JUnit test runner.

3. Formal Models: Several formal models have been proposed to add semantic specification directly to the programming language, usually following a variant

of the "design by contract" approach (Mitchell et al., 2002). One such formal model to define the behavioural specification is the Java Modelling Language (JML) (Burdy et al., 2003). In JML, behaviour specifications are added as comments to the source code and can be verified with the aid of several existing tools such as extended specification checker for Java (ESC/Java) (Flanagan et al., 2002). Programs can be verified against JML specifications using static and behavioural techniques.

In the context of these verification methods, we cannot completely asses the correctness of programs due to lack of formal models and tests for all programs in the dataset. None of the programs had JML style annotations. Test cases were available for 28 programs. We have manually tested 5 programs before and after refactoring because these programs have the highest test coverage among other programs. The test coverage for the selected programs ranged between 45 - 85%.

On the other hand, the only model we always have is the type system usage by the program, verified by the Java compiler. The type system checking cannot always prove correctness. There can be situations where our models do not correctly represent the code. For example, the move class refactoring requires that all references to the move class must be updated. In some cases, class names may be hard-coded in external configuration files and then loaded via reflection at runtime. In case of the move class refactoring, the refactored program will compile successfully, however we may get a runtime exception if the class name in the external configuration file is not updated. This may influence the outcome in a way that some of the refactorings could not be successful due to the postcondition failure. These cases require different models to check postconditions such as behavioural analysis using test cases or static analysis that includes these configuration files. For instance, using framework specific profiles that include standard configuration files like J2EE deployment descriptors.

6.2.3 Developers Feedback

There are other models to verify the correctness of programs that are not technical but rather subjective. For example, organisations develop coding styles for their projects. It is possible that refactorings may violate those guidelines. Similarly, there are other quality attributes and metrics that are specifically designed and developed for projects. The only way to find out about those is the developer feedback. In our study, we have focused on a large number of open source systems that were produced by many developers around the globe. We didn't involve developers in the process because it is a time consuming activity and the response to surveys and questionnaires is usually very low. Instead, we focused on the improvement of fitness functions as a criterion to the improved quality. However, it would be interesting to involve developers in the process and get their feedback on the suggested refactorings. This would provide us with another notion of correctness of refactored programs, similar to section 6.2.2.

6.2.4 Influence of Tools

Eclipse Juno 4.2 has been used for the proof-of-concept implementation. We have used several Eclipse APIs to develop refactorings, such as Java Development Tool (JDT) and Refactoring Language Toolkit (LTK). We also used some of the standard Eclipse refactorings including move refactoring, introduce factory refactoring and generalize declared type refactoring. We chose Eclipse because it has well documented APIs and a very active community. However, some of the refactorings we used have known bugs and limitations, which are discussed in section 4.6.11. There are several other tools that provide Abstract Syntax Tree (AST) APIs to manipulate the source code of programs. These tools include IntelliJ IDEA, NetBeans, and RECODER[1]. We believe that the implementation tools may have had some influence on the outcome of results. We expect the impact to be minor since other tools might have their own flaws, which are unknown to us.

6.2.5 Java Specific Refactorings

In this research, we have investigated the automation of architectural refactorings of Java programs. Our approach is heavily based on static analysis, and therefore depends on static typing. We expect that these results can be generalised to other languages with a similar type system. It is not clear how it can be used with dynamically typed languages, such as Smalltalk. Further investigation is required in the area of language independent refactorings. For example, Tichelaar et al. (2000) developed a language-independent meta-model called FAMIX, which represents programs written in Java, Smalltalk, and other object-oriented languages on an abstract level. This representation is based on an entity relationship model. The core of this model has three main entities: Class, Method, and Attribute and relationships between them. The FAMIX model is implemented through Moose refactoring engine, which contains several language independent primitive refactorings including move and rename refactorings. With the help of such tools, it is possible to write language-independent refactorings, such as move classes and move methods to remove some of the antipatterns discussed in this work. The other types of refactorings such as type generalisation and service

[1]`http://sourceforge.net/projects/recoder/`

locators may not be required in dynamically typed languages because these refactorings try to resolve problems created due to static typing constraints. This implies that certain architectural antipatterns disappear from programs. For example, Smalltalk doesn't support multiple inheritance through interfaces. Therefore, the degenerated inheritance antipattern would not be present in programs written in this language.

6.2.6 Scalability

Since we apply refactorings on a large set of programs, scalability becomes a big challenge when dealing with large programs. For example, keeping the AST of all classes of a program in the memory is a computationally intensive task and may cause heap problems. To deal with that problem, we loaded only the ASTs of required classes (source and target classes of a dependency) in a program. The scalability issues may arise if we weaken our preconditions. For example, if we allow type generalisation refactoring to follow leaking references (Section 5.4.2.1), ASTs of referenced classes must be loaded into the memory recursively. However, this may influence the results so that more type generalisation refactorings might be successful.

6.3 Research Contributions

The thesis makes three contributions. First, it demonstrates with an empirical study, using a large corpus of real-world programs that certain architectural refactorings can be safely automated to some extent. To our knowledge, this is a unique study where the source code of such a large number of programs was refactored automatically.

Second, several novel algorithms have been developed that automatically translate the model refactorings to code refactorings. The existing approaches identified refactorings on the model level while our approach provides a complete solution, i.e., detecting critical dependencies on the model level, selecting appropriate refactorings, and applying refactorings safely on the source code level.

Third, we have implemented our algorithms as Eclipse plugins. We have designed our refactoring framework to allow the execution of composite refactorings, whereas the existing tools do not allow the execution of composite refactorings. The composite refactoring is a combination of several primitive refactorings. Due to the complexity of code, sometimes we need several refactorings to break a dependency from source to target. We have developed a transactional mechanism for refactorings. This allowed us to roll back a composite refactoring in the event of postcondition failure. Moreover, our plugins perform the complete refactoring cycle with a push of a button. All the refactoring related information is recorded for the review purpose. In addition, the plugin implementation can analyse and refactor programs in a batch-mode.

6.4 Future Work

Our research is an initial step towards the automation of architectural refactorings. This is a complex task, which requires further research to explore the benefits of automation. The current work comprises of a particular set of antipatterns that can be extended to include project specific antipatterns or any other design violations. Similarly, we have a specific number of refactoring techniques to break dependencies between classes. Other refactoring techniques such as the introduction of dependency injection can be added to the current set of refactorings to narrow down the gap between refactoring simulation and refactoring application.

In addition, the results can be improved by the development of robust primitive code level refactorings. Our architectural refactoring framework is built on the standard code level refactorings. For composite refactorings to be successful, it is important to have robust and safe primitive refactorings. For example, in section 4.6.11, several limitations of the move refactoring have been described, which prevent the successful execution of the relevant refactoring.

Further improvements can be made in the area of pre and postconditions for refactorings. We had several relatively strong preconditions, which can be weakened to achieve better success rates. For example, while performing type generalisation refactoring, we didn't generalise a variable type when it was passed to another class via a method invocation (see Section 5.4.2.1). This can be allowed by recursively following the leaked references. Similarly, a move refactoring was not performed where the change visibility refactoring was required.

Moreover, refactoring postconditions had basic safeguard conditions, i.e., after a refactoring, the instance count metric should not increase and the program compilation should succeed. Although, we manually executed test cases of a few programs to check the correctness of the program before and after refactoring, it is also important to check testcases and other design quality metrics after each refactoring. Postconditions can be improved by adding the successful execution of build script after each refactoring. This would allow the refactoring tool to check several postconditions such as successful compilation, test cases, coupling and cohesion metrics and so on.

Bibliography

Abdeen, H., Ducasse, S., Pollet, D., and Alloui, I. (**2010**). "Package Fingerprints: A visual summary of package interface usage." *Inf. Softw. Technol.*, 52: 1312–1330.

Abdeen, H., Ducasse, S., Sahraoui, H. A., and Alloui, I. (**2009**). "Automatic Package Coupling and Cycle Minimization." In "WCRE'09," pages 103–112.

Anson, M., Fabozzi, F., and Jones, F. (**2010**). *The Handbook of Traditional and Alternative Investment Vehicles: Investment Characteristics and Strategies*. Frank J. Fabozzi Series. Wiley. Retrieved from `http://books.google.co.nz/books?id=LKj39XK-ufsC`.

Bach, M., Forster, F., and Steimann, F. (**2007**). "Declared Type Generalization Checker: An Eclipse Plug-In for Systematic Programming with More General Types." In M. B. Dwyer and A. Lopes, editors, "FASE 2007," volume 4422 of *LNCS*, pages 117–120. Springer.

Bajracharya, S., Ngo, T., Linstead, E., Rigor, P., Dou, Y., Baldi, P., and Lopes, C. (**2006**). "Sourcerer: A search engine for open source code supporting structure-based search." In "In Proc. Intl Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA06," pages 25–26.

Bansiya, J. and Davis, C. G. (**2002**). "A Hierarchical Model for Object-Oriented Design Quality Assessment." *IEEE Trans. Softw. Eng.*, 28: 4–17.

Beck (**2002**). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ben Asher, Y., Gal, T., Haber, G., and Zalmanovici, M. (**2012**). "Refactoring techniques for aggressive object inlining in Java applications." *Automated Software Engineering*, 19: 97–136. doi:10.1007/s10515-011-0096-x.

Beyer, D. and Lewerentz, C. (**2003**). "CrocoPat: A Tool for Efficient Pattern Recognition in Large Object-Oriented Programs." Technical report, -.

Bischofberger, W. R., Khl, J., and Lffler, S. (**2004**). "Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking." In "EWSA," volume 3047 of *LNCS*, pages 1–9. Springer.

Blewitt, A. (**2009**). "Modular Java: What Is It?" Retrieved from `http://www.infoq.com/articles/modular-java-what-is-it`.

Brin, S. and Page, L. (**1998**). "The anatomy of a large-scale hypertextual Web search engine." *Comput. Netw. ISDN Syst.*, 30(1-7): 107–117.

Brooks, F. P., Jr. (**1987**). "No Silver Bullet Essence and Accidents of Software Engineering." *Computer*, 20(4): 10–19. doi:10.1109/MC.1987.1663532.

Brown, W. J., Malveau, R. C., McCormick, H. W., III, and Mowbray, T. J. (**1998**). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. (**2003**). "An overview of JML tools and applications." Retrieved from `citeseer.ist.psu.edu/burdy03overview.html`.

Burn, O. (**2008**). "Checkstyle." Retrieved from `http://checkstyle.sourceforge.net/`.

Clark, M. (**2003**). "JDepend Dependency Analyser." Retrieved from `http://clarkware.com/software/JDepend.html`.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (**1994**). "Using Metrics to Evaluate Software System Maintainability." *Computer*, 27(8): 44–49. doi:10.1109/2.303623.

Copeland, T. (**2005**). *PMD Applied*. Centennial Books.

Czibula, I. and Serban, G. (**2007**). "Hierarchical Clustering for Software Systems Restructuring." *INFOCOMP, Journal of Computer Science*, 6(4): 43 – 51.

Da Vinci Machine (**2008**). "the Da Vinci Machine Project Homepage." Retrieved from `http://openjdk.java.net/projects/mlvm/`.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (**2009**). *Object-Oriented Reengineering Patterns*. Square Bracket Associates.

Dietrich, J. (**2012**). "Upload your program, share your model." In "Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity," SPLASH '12, pages 21–22. ACM, New York, NY, USA. doi:10.1145/2384716.2384727.

Dietrich, J. (**2013**). "Orthogonality by example: The principles of modular and maintainable design in Log4j." Retrieved from `http://www.javaworld.com/javaworld/jw-05-2013/130501-jtip-orthogonality-by-example.html`.

Dietrich, J. and McCartin, C. (**2012**). "Scalable Motif Detection and Aggregation." In "Australasian Database Conference (ADC 2012)," volume 124 of *CRPIT*, pages 31–40. ACS, Melbourne, Australia. Retrieved from `http://crpit.com/confpapers/CRPITV124Dietrich.pdf`.

Dietrich, J., McCartin, C., Tempero, E., and Shah, S. M. A. (**2010**). "Barriers to modularity: an empirical study to assess the potential for modularisation of java programs." In "Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps," QoSA'10, pages 135–150. Springer-Verlag, Berlin, Heidelberg.

Dietrich, J., McCartin, C., Tempero, E., and Shah, S. M. A. (**2012**). "On the existence of high-impact refactoring opportunities in programs." In "Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122," ACSC '12, pages 37–48. Australian Computer Society, Inc., Darlinghurst, Australia, Australia. Retrieved from `http://dl.acm.org/citation.cfm?id=2483654.2483659`.

Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., and Duchrow, M. (**2008**). "Cluster analysis of Java dependency graphs." In R. Koschke, C. D. Hundhausen, and A. Telea, editors, "Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008," pages 91–94. ACM.

Do, H., Elbaum, S., and Rothermel, G. (**2005**). "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." In "-," volume 10, pages 405–435. Kluwer Academic Publishers, Hingham, MA, USA.

Dolby, J. (**1997**). "Automatic inline allocation of objects." In "Proceedings of the ACM SIGPLAN 1997 conference on PLDI'97," PLDI '97, pages 7–17. ACM, New York, NY, USA.

Dolby, J. and Chien, A. A. (**2000**). "An Automatic Object Inlining Optimization and its Evaluation." In "In PLDI 2000," pages 345–357. ACM Press.

Eades, P., Lin, X., and Smyth, W. (**1993**). "A fast and effective heuristic for the feedback arc set problem." *Information Processing Letters*, 47(6): 319 – 323.

Falleri, J.-R., Denier, S., Laval, J., Vismara, P., and Ducasse, S. (**2011**). "Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems." In "TOOLS," Zurich, Suisse.

Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). "Extended static checking for Java." In "Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation," PLDI '02, pages 234–245. ACM, New York, NY, USA. doi:10.1145/512529.512558.

Foote, B. and Yoder, J. (1997). "Big Ball of Mud." In "Pattern Languages of Program Design," pages 653–692. Addison-Wesley.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Fowler, M. (2001). "Reducing Coupling." Retrieved from `http://martinfowler.com/ieeeSoftware/coupling.pdf`.

Fowler, M. (2004). "Inversion of Control Containers and the Dependency Injection pattern." Retrieved from `http://martinfowler.com/articles/injection.html#InversionOfControl`.

Freeman, L. C. (1977). "A set of measures of centrality based on betweenness." *Sociometry*, 40(1): 35–41.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA.

Girvan, M. and Newman, M. E. J. (2002). "Community structure in social and biological networks." *Proceedings of the National Academy of Sciences*, 99(12): 7821–7826.

Gossner, J., Mayer, P., and Steimann, F. (2004). "Interface utilization in the Java Development Kit." In "Proceedings of the 2004 ACM symposium on Applied computing," SAC '04, pages 1310–1315. ACM, New York, NY, USA.

Grady, R. (1994). "Successfully applying software metrics." *Computer*, 27(9): 18 –25.

Grothoff, C., Palsberg, J., and Vitek, J. (2007). "Encapsulating objects with confined types." *ACM Trans. Program. Lang. Syst.*, 29(6). doi:10.1145/1286821.1286823.

Hammant, P. and Hellesy, A. (2003). "Pico Container." Retrieved from `http://www.picocontainer.org/`.

Harmony (2010). "Apache Harmony." Retrieved from `http://harmony.apache.org/`.

Hautus, E. (2002). "Improving java software through package structure analysis." In "IASTED International Conference Software Engineering and Applications," .

Hovemeyer, D. and Pugh, W. (**2004**). "Finding bugs is easy." In "OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications," pages 132–136. ACM, New York, NY, USA.

Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., and Kopylenko, D. (**2005**). *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK.

Knoernschild, K. (**2012**). *Java Application Architecture: Modularity Patterns With Examples Using Osgi*. Agile Software Development Series. Prentice Hall. Retrieved from `http://books.google.co.nz/books?id=iOtwFoU1Dt4C`.

Koenig, A. (**1998**). "The patterns handbooks." chapter Patterns and antipatterns, pages 383–389. Cambridge University Press, New York, NY, USA. Retrieved from `http://dl.acm.org/citation.cfm?id=301570.301985`.

Lakos, J. (**1996**). *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Laval, J., Denier, S., Ducasse, S., and Bergel, A. (**2009**). "Identifying Cycle Causes with Enriched Dependency Structural Matrix." In "Reverse Engineering, 2009. WCRE '09. 16th Working Conference on," pages 113 –122.

Lehman, M. (**1979**). "On understanding laws, evolution, and conservation in the large-program life cycle." *Journal of Systems and Software*, 1(0): 213 – 221. doi:10.1016/0164-1212(79)90022-0.

Lethbridge, T. and Laganiere, R. (**2002**). *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Lippert, M. and Roock, S. (**2006**). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley.

Madadhain, J., Fisher, D., Smyth, P., White, S., and Boey, Y. (**2005**). "Analysis and visualization of network data using JUNG." *Journal of Statistical Software*, 10: 1–35.

Mancoridis, S., Mitchell, B., Rorres, C., Chen, Y., and Gansner, E. (**1998**). "Using automatic clustering to produce high-level system organizations of source code." In "Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on," pages 45 –52.

Maqbool, O. and Babri, H. (**2007**). "Hierarchical Clustering for Software Architecture Recovery." *Software Engineering, IEEE Transactions on*, 33(11): 759 –780.

Martin, R. (**1994**). "Object oriented design quality metrics: An analysis of dependencies." Report on object analysis and design. Retrieved from `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`.

Martin, R. (**2000**). "Design Principles and Design Patterns." Retrieved from `http://www.objectmentor.com`.

Martin, R. C. (**2003**). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Matuszak, A. (**2010**). "Differences between Arithmetic, Geometric, and Harmonic Means." Retrieved from `http://economistatlarge.com/finance/applied-finance/differences-arithmetic-geometric-harmonic-means`.

Mayer, P. (**2003**). "Analyzing the use of interfaces in large OO projects." In "Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications," OOPSLA '03, pages 382–383. ACM, New York, NY, USA.

Mayer, P., Meissner, A., and Steimann, F. (**2007**). "A visual interface for type-related refactorings." In "WRT'07," pages 5–6.

Melton, H. and Tempero, E. (**2006**). "Identifying refactoring opportunities by identifying dependency cycles." In "Proceedings of the 29th Australasian Computer Science Conference - Volume 48," ACSC '06, pages 35–41. Australian Computer Society, Inc., Darlinghurst, Australia, Australia.

Melton, H. and Tempero, E. (**2007**a). "An empirical study of cycles among classes in Java." *Empirical Softw. Engg.*, 12: 389–415.

Melton, H. and Tempero, E. D. (**2007**b). "An empirical study of cycles among classes in Java." *Empirical Software Engineering*, 12(4): 389–415.

Miller, J. C. and Maloney, C. J. (**1963**). "Systematic mistake analysis of digital computer programs." *Commun. ACM*, 6(2): 58–63. doi:10.1145/366246.366248.

Mitchell, B. and Mancoridis, S. (**2006**). "On the automatic modularization of software systems using the Bunch tool." *Software Engineering, IEEE Transactions on*, 32(3): 193 – 208.

Mitchell, R., McKim, J., and Meyer, B. (**2002**). *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Muchnick, S. S. (**1997**). *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

MVEL (**2009**). "MVEL Expression Language." Retrieved from `http://mvel.codehaus.org/`.

O'Keeffe, M. and O'Cinneide, M. (**2006**). "Search-Based Software Maintenance." *Software Maintenance and Reengineering, European Conference on*, 0: 249–260.

Opdyke, W. (**1992**). *Refactoring object-oriented frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA.

Opdyke, W. F. and Johnson, R. E. (**1993**). "Creating abstract superclasses by refactoring." In "Proceedings of the 1993 ACM conference on Computer science," CSC '93, pages 66–73. ACM, New York, NY, USA.

Osherove, R. (**2009**). *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Parnas, D. L. (**1972**). "On the criteria to be used in decomposing systems into modules." *Commun. ACM*, 15(12): 1053–1058.

Pek, E. and Lämmel, R. (**2013**). "A Literature Survey on Empirical Software Engineering Research."

Pierce, B. C. (**2002**). *Types and programming languages*. MIT Press, Cambridge, MA, USA.

Project Jigsaw (**2008**). "The Project Jigsaw Homepage." Retrieved from `http://openjdk.java.net/projects/jigsaw/`.

RefactoringCatalog (**1999**). "Move Class Refactoring." Retrieved from `http://refactoring.com/catalog/index.html`.

Reinhold, M. (**2012**). "Project Jigsaw: Late for the train: The Q&A." Retrieved from `http://mreinhold.org/blog/late-for-the-train-qa`.

Research and Markets (**2011**). "Software: Global Industry Guide (Abstract)." Retrieved from `http://www.researchandmarkets.com/reports/2064382/software_global_industry_guide`.

ReStructure101 (**2008**). "ReStructure101, Headway Software Technologies." Retrieved from `http://www.headwaysoftware.com/products/?code=Restructure101`.

Riel, A. J. (**1996**). *Object-Oriented Design Heuristics*. Addison-Wesley, Boston, MA, USA.

Sakkinen, M. (**1989**). "Disciplined Inheritance." *ECOOP'89: Proceedings of the 1989 European Conference on Object-Oriented Programming*, -: 39–56.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (**2005**). "Using dependency models to manage software architecture." In "OOPSLA '05," pages 164–165. ACM, New York, NY, USA.

Schach, S. R. (**1996**). *Software Engineering with Java*. McGraw-Hill Professional.

Seng, O., Bauer, M., Biehl, M., and Pache, G. (**2005**). "Search-based improvement of subsystem decompositions." In "Proceedings of the 2005 conference on Genetic and evolutionary computation," GECCO '05, pages 1045–1051. ACM, New York, NY, USA.

Shah, S. M. A., Dietrich, J., and McCartin, C. (**2012**). "On the automated modularisation of java programs using service locators." In "Proceedings of the 11th international conference on Software Composition," SC'12, pages 132–147. Springer-Verlag, Berlin, Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-30564-1_9`.

Stal, M. (**2008**). "Refactoring of Software Architecture." OOPSLA 2008. Retrieved from `http://www.oopsla.org/oopsla2008/tutorials.html`.

Steimann, F., Mayer, P., and MeiBner, A. (**2006**). "Decoupling classes with inferred interfaces." In "Proceedings of the 2006 ACM symposium on Applied computing," SAC '06, pages 1404–1408. ACM, New York, NY, USA.

Steimann, F., Siberski, W., and Kuhne, T. (**2003**). "Towards the systematic use of interfaces in JAVA programming." In "Proceedings of the 2nd international conference on Principles and practice of programming in Java," PPPJ '03, pages 13–17. Computer Science Press, Inc., New York, NY, USA.

Stevens, W., Myers, G., and Constantine, L. (**1979**). *Structured design*, pages 205–232. Yourdon Press, Upper Saddle River, NJ, USA.

Streckenbach, M. and Snelting, G. (**2004**). "Refactoring class hierarchies with KABA." In "Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications," OOPSLA '04, pages 315–330. ACM, New York, NY, USA.

Taube-Schock, C., Walker, R. J., and Witten, I. H. (**2011**). "Can We Avoid High Coupling?" In "ECOOP," pages 204–228.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (**2010**). "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies." *Proc. APSEC*, -: –.

Tempero, E. D. (**2008**). "An Empirical Study of Unused Design Decisions in Open Source Java Software." In "APSEC," pages 33–40. IEEE.

Tessier, J. (**2010**). "Dependency Finder." Retrieved from `http://depfind.sourceforge.net/`.

Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (**2000**). "A meta-model for language-independent refactoring." In "Principles of Software Evolution, 2000. Proceedings. International Symposium on," pages 154–164.

Tip, F., Kiezun, A., and Bäumer, D. (**2003**). "Refactoring for generalization using type constraints." In "Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications," OOPSLA '03, pages 13–26. ACM, New York, NY, USA.

Tsantalis, N. (**2007**). "Bad Smell Identification for Software Refactoring." Retrieved from `http://www.jdeodorant.org`.

Tsantalis, N. and Chatzigeorgiou, A. (**2009**). "Identification of Move Method Refactoring Opportunities." *IEEE Transactions on Software Engineering*, 99(RapidPosts): 347–367.

Vanbrabant, R. (**2008**). *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress.

Walls, C. (**2009**). *Modular Java: Creating Flexible Applications with Osgi and Spring*. Pragmatic Bookshelf.

Yourdon, E. (**1992**). *Decline and fall of the American programmer*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Zilberstein, S. (**1996**). "Using Anytime Algorithms in Intelligent Systems." *AI Magazine*, 17(3): 73–83. Retrieved from `http://rbr.cs.umass.edu/shlomo/papers/Zaimag96.html`.

Appendix A

Declaration of Previous Work

Parts of this thesis are based on our previously published work as follows:

- **Dietrich, J., McCartin, C., Tempero, E., and Shah, S. M. A. (2010)**. "Barriers to modularity: an empirical study to assess the potential for modularisation of Java programs." In "Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps," QoSA '10, pages 135-150. Springer-Verlag, Berlin, Heidelberg.
- **Dietrich, J., McCartin, C., Tempero, E., and Shah, S. M. A. (2012)**. "On the existence of high-impact refactoring opportunities in programs." In "Proceedings of the Thirty fifth Australasian Computer Science Conference - Volume 122," ACSC '12, pages 37-48. Australian Computer Society, Inc., Darlinghurst, Australia.
- **Shah, S. M. A., Dietrich, J., and McCartin, C. (2012)**. "Making smart moves to untangle programs." In "Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering," CSMR '12, pages 359-364. IEEE Computer Society, Washington, DC, USA.
- **Shah, S. M. A., Dietrich, J., and McCartin, C. (2012)**. "On the automated modularisation of Java programs using service locators." In "Proceedings of the 11th international conference on Software Composition," SC '12, pages 132-147. Springer-Verlag, Berlin, Heidelberg.
- **Shah, S. M. A., Dietrich, J., and McCartin, C. (2013)**. "On the automation of dependency-breaking refactorings in Java." In "Proceedings of the 29th IEEE International Conference on Software Maintenance," ICSM '13, pages 160-169. IEEE Computer Society, Washington, DC, USA.

Appendix B

CARE Plugin: Installation and Instructions

B.1 Installation

The CARE plugin can be installed via Eclipse IDE using **Help** → **Install New Software** option, as shown in figure B.1. The following update site can be used to install the plugin for Eclipse 4.0.x to Eclipse 4.2.x.

CARE update site: `http://care.googlecode.com/svn/trunk/care-update-site/`

B.1.1 Configuration

Some large projects may cause Heap size OutOfMemoryError. In order to avoid the Heap size problem, update the eclipse.ini file with the following parameters:

```
-vmargs
-Xms512m
-Xmx2048m
-XX:PermSize=256m
```

The **eclipse.ini** file on **Mac OS X** can be accessed by right-clicking the Eclipse application icon and then clicking **Show Package Contents**. This will open a new Finder window with the **Contents** folder. In this folder, open **MacOS** folder. The **eclipse.ini** file can be found here.

B.2 Usage Instructions

B.2.1 User Interface

Once the plugin is installed, it can be accessed through **CARE** menu (added to the Eclipse workbench) where we can select different refactoring options. Currently, programs can be restructured in two different ways.

1. Restructure using Move Class Refactoring.
2. Restructure using Composite Refactoring.

The composite refactoring technique comprises of four types of refactorings including move class, type generalization, introduction of service locators, and static members inlining.

The user interface of the CARE plugin is shown in figure B.2. In order to start the refactoring process, programs must be loaded into the Eclipse workspace. The refactoring process of a single project can be started by clicking on the left button of the selection at the top right hand side of the refactoring view panel, as shown in the figure. This will open a dialog box. Next, we can select a project from the dialog box and the refactoring process will begin. To analyse all opened projects in the workspace, click on the right button (Analyse All Projects).

B.2.2 Preferences

Some options can be configured in the CARE plugin. For example, we can edit the maximum number of refactoring steps for a program. The default value is 10. In a similar way, we can pre load certain class names that should not be moved during the automated refactoring process, as shown in figure B.3.

B.2.3 Importing Projects

As described in section 2.6, we had manually configured programs in the Qualitas Corpus as Eclipse projects. These projects can be downloaded from the project website[1]. All programs are configured to the folder structure as shown in table B.1.

Table B.2 shows two files that can be download from the project website. These are zip files and contain pre-configured Eclipse projects.

[1]`https://code.google.com/p/care/wiki/Documentation`

Folder	Description
src	Source code contained in .java files goes to this folder.
bin	The compiled output files (.class) are placed here.
lib	This folder contains external libraries required by a project. Note that external libraries are excluded from the refactoring process.
tests	Source code of test cases is placed under this folder, if provided.

Table B.1: Eclipse Project Structure

File Name	No. of Programs	File Size
sample-programs.zip	5	15 MB
corpus-20101126r.zip	92	1 GB

Table B.2: Dataset Files

After downloading the dataset file, use the **File** → **Import** option in the Eclipse IDE to load these programs in a workspace. Before importing projects, the following requirements must be fulfilled:

1. JDK 1.6.x must be installed on the system because all projects were configured and compiled using this JDK. Choosing a different JDK version (1.5 or 1.7) may require a clean build for all projects.
2. Under Eclipse Preferences, click on **Java** → **Compiler** → **Errors/Warnings**. In the section **Deprecated and Restricted APIs**, choose **Ignore** for **Forbidden Reference (access rule)**. For example, in JMoney-0.4.4, the class `net.sf.jmoney.gui.-DateComboBox` uses `com.sun.java.swing.plaf.motif.WindowsComboBoxUI`. The `WindowsComboBoxUI` class is part of `classes.jar` (JDK on MacOS), whose access is restricted by default. Therefore, a compilation error is generated. Choosing **Ignore** for **Forbidden Reference** would resolve the problem.

Using the **File** → **Import** option, projects can be loaded into the workspace, as shown in the figures B.4 and B.5. After the import process if some projects show compilation errors, select those projects and clean them using the **Project** → **Clean** option.

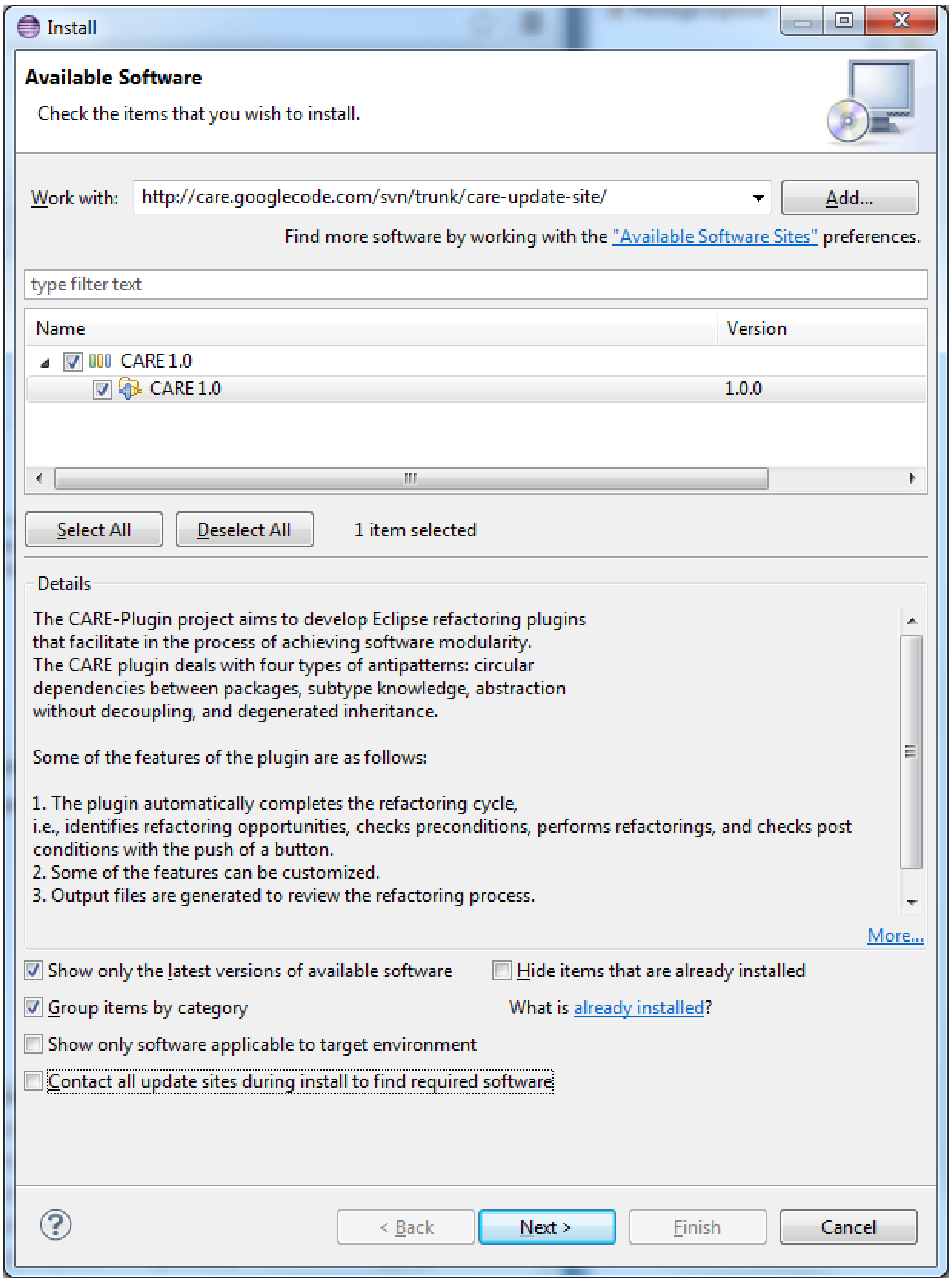
B.2.4 Refactoring Output

After the refactoring process is completed, we can view the refactoring related information in the output folder of each project. The output is generated as comma separated (CSV) files. The description of these files for the move class refactoring is shown in table B.3.

File name	Description
instances	Keeps track of antipattern instance count after each refactoring.
metrics	All other metrics are stored in this file, such as strongly connected components metrics.
g2c-success	Graph-to-Code success rate. This data is used to compute the refactorability metric.
constraints	Information related to pre and postconditions is stored here.
details	This file contains detailed information about the removed dependency, such as how many patterns were removed.
error-edge	Due to limitations of Eclipse refactorings, some refactorings may not be rolled back properly, thus resulting in the program compilation failure. This file contains the name of class that created the problem. We can put this class name in the list of blacklisted classes in Eclipse Preferences. For example, we found that in JHotdraw-7.5.1, when the class `org.jhotdraw.draw.io.-InputFormat` is attempted to move into a different package, this results in compilation errors due to invalid imports (described in Section 4.6.11). Therefore, in order to avoid the compilation problem, we can add this class name to blacklist.
package-metrics	Here all the package related metrics are stored including before and after values.

Table B.3: Output Description

The output files for the composite refactoring are generated under the folder named **output-composite-refactoring**. Here, a sub-folder **refactored-code** exists, which contains source code before and after refactoring for each successful refactoring (excluding the move class refactoring).


Install
Available Software
Check the items that you wish to install.

Work with: http://care.googlecode.com/svn/trunk/care-update-site/

Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text

Name	Version
☑ CARE 1.0	
☑ CARE 1.0	1.0.0

1 item selected

Details

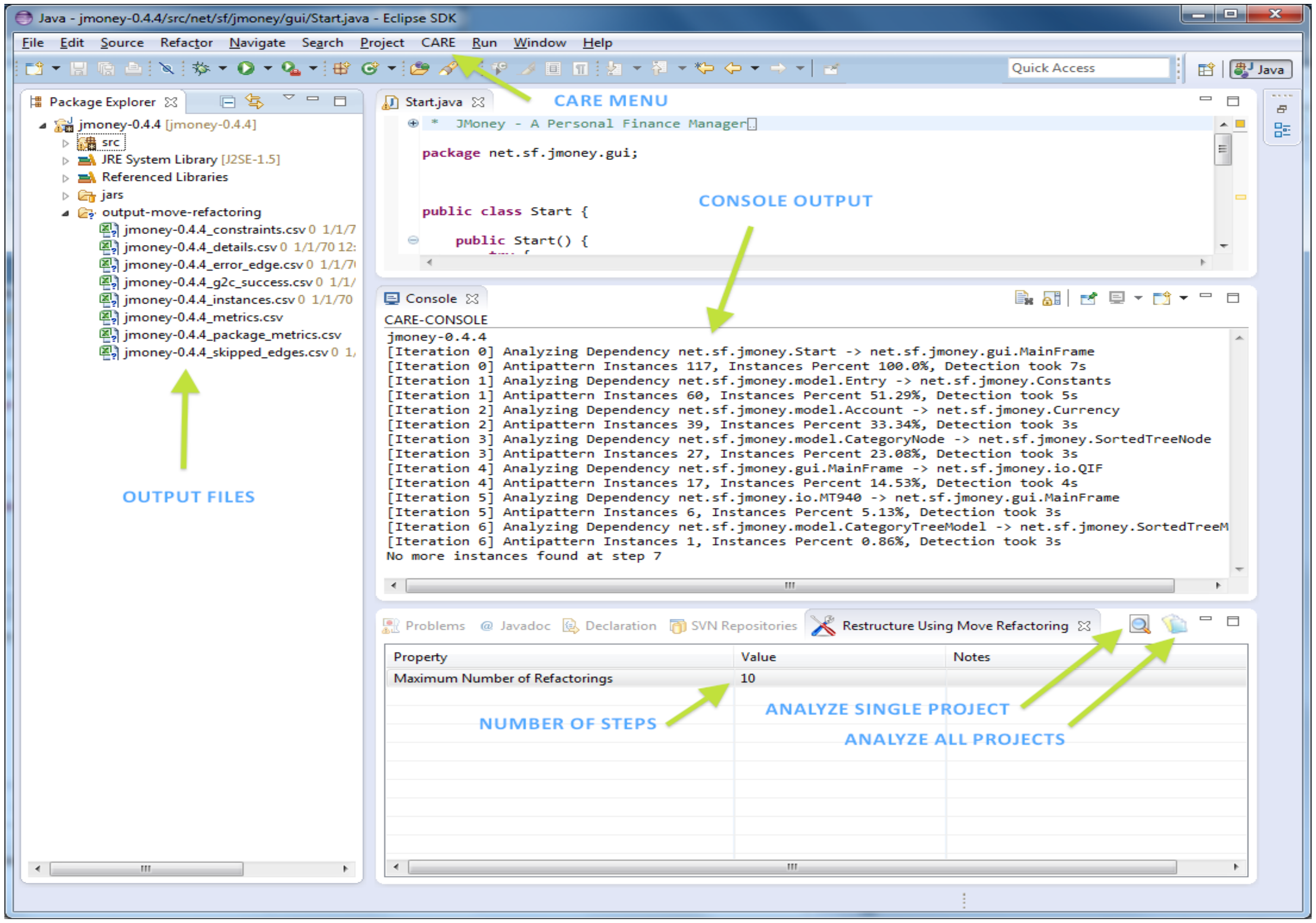
The CARE-Plugin project aims to develop Eclipse refactoring plugins that facilitate in the process of achieving software modularity.
The CARE plugin deals with four types of antipatterns: circular dependencies between packages, subtype knowledge, abstraction without decoupling, and degenerated inheritance.

Some of the features of the plugin are as follows:

1. The plugin automatically completes the refactoring cycle, i.e., identifies refactoring opportunities, checks preconditions, performs refactorings, and checks post conditions with the push of a button.
2. Some of the features can be customized.
3. Output files are generated to review the refactoring process.

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☐ Contact all update sites during install to find required software

Figure B.1: CARE Installation

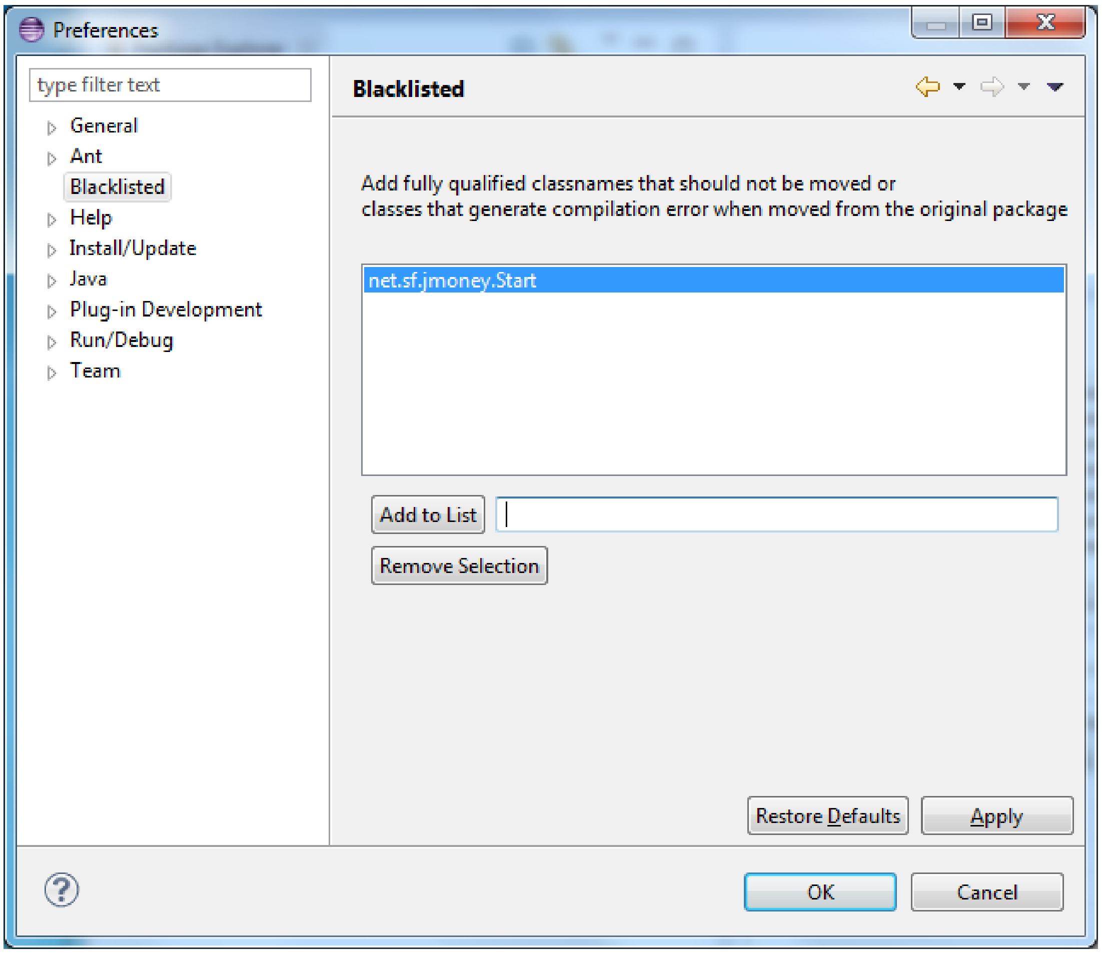

Java - jmoney-0.4.4/src/net/sf/jmoney/gui/Start.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project CARE Run Window Help
Quick Access
Java
Package Explorer
jmoney-0.4.4 [jmoney-0.4.4]
src
JRE System Library [J2SE-1.5]
Referenced Libraries
jars
output-move-refactoring
jmoney-0.4.4_constraints.csv 0 1/1/7
jmoney-0.4.4_details.csv 0 1/1/70 12:
jmoney-0.4.4_error_edge.csv 0 1/1/7
jmoney-0.4.4_g2c_success.csv 0 1/1/
jmoney-0.4.4_instances.csv 0 1/1/70
jmoney-0.4.4_metrics.csv
jmoney-0.4.4_package_metrics.csv
jmoney-0.4.4_skipped_edges.csv 0 1,
OUTPUT FILES
Start.java
CARE MENU
CONSOLE OUTPUT
Console
Problems
Javadoc
Declaration
SVN Repositories
Restructure Using Move Refactoring
NUMBER OF STEPS
ANALYZE SINGLE PROJECT
ANALYZE ALL PROJECTS

```
 * JMoney - A Personal Finance Manager

package net.sf.jmoney.gui;


public class Start {

    public Start() {
```

```
CARE-CONSOLE
jmoney-0.4.4
[Iteration 0] Analyzing Dependency net.sf.jmoney.Start -> net.sf.jmoney.gui.MainFrame
[Iteration 0] Antipattern Instances 117, Instances Percent 100.0%, Detection took 7s
[Iteration 1] Analyzing Dependency net.sf.jmoney.model.Entry -> net.sf.jmoney.Constants
[Iteration 1] Antipattern Instances 60, Instances Percent 51.29%, Detection took 5s
[Iteration 2] Analyzing Dependency net.sf.jmoney.model.Account -> net.sf.jmoney.Currency
[Iteration 2] Antipattern Instances 39, Instances Percent 33.34%, Detection took 3s
[Iteration 3] Analyzing Dependency net.sf.jmoney.model.CategoryNode -> net.sf.jmoney.SortedTreeNode
[Iteration 3] Antipattern Instances 27, Instances Percent 23.08%, Detection took 3s
[Iteration 4] Analyzing Dependency net.sf.jmoney.gui.MainFrame -> net.sf.jmoney.io.QIF
[Iteration 4] Antipattern Instances 17, Instances Percent 14.53%, Detection took 4s
[Iteration 5] Analyzing Dependency net.sf.jmoney.io.MT940 -> net.sf.jmoney.gui.MainFrame
[Iteration 5] Antipattern Instances 6, Instances Percent 5.13%, Detection took 3s
[Iteration 6] Analyzing Dependency net.sf.jmoney.model.CategoryTreeModel -> net.sf.jmoney.SortedTreeM
[Iteration 6] Antipattern Instances 1, Instances Percent 0.86%, Detection took 3s
No more instances found at step 7
```

Property	Value	Notes
Maximum Number of Refactorings	10	

Figure B.2: CARE User Interface



Blacklisted

Add fully qualified classnames that should not be moved or
classes that generate compilation error when moved from the original package

net.sf.jmoney.Start

Figure B.3: CARE Preferences

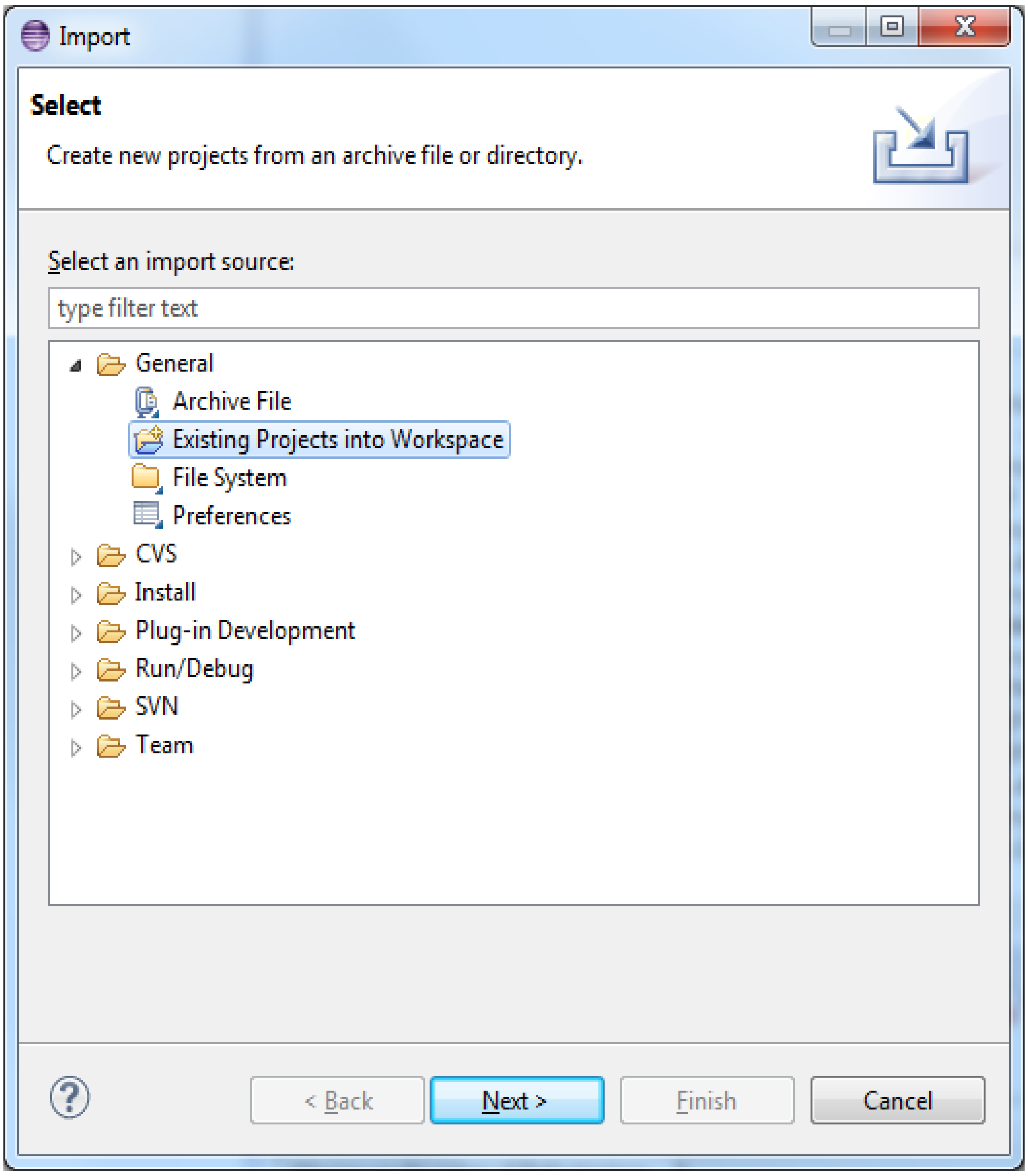
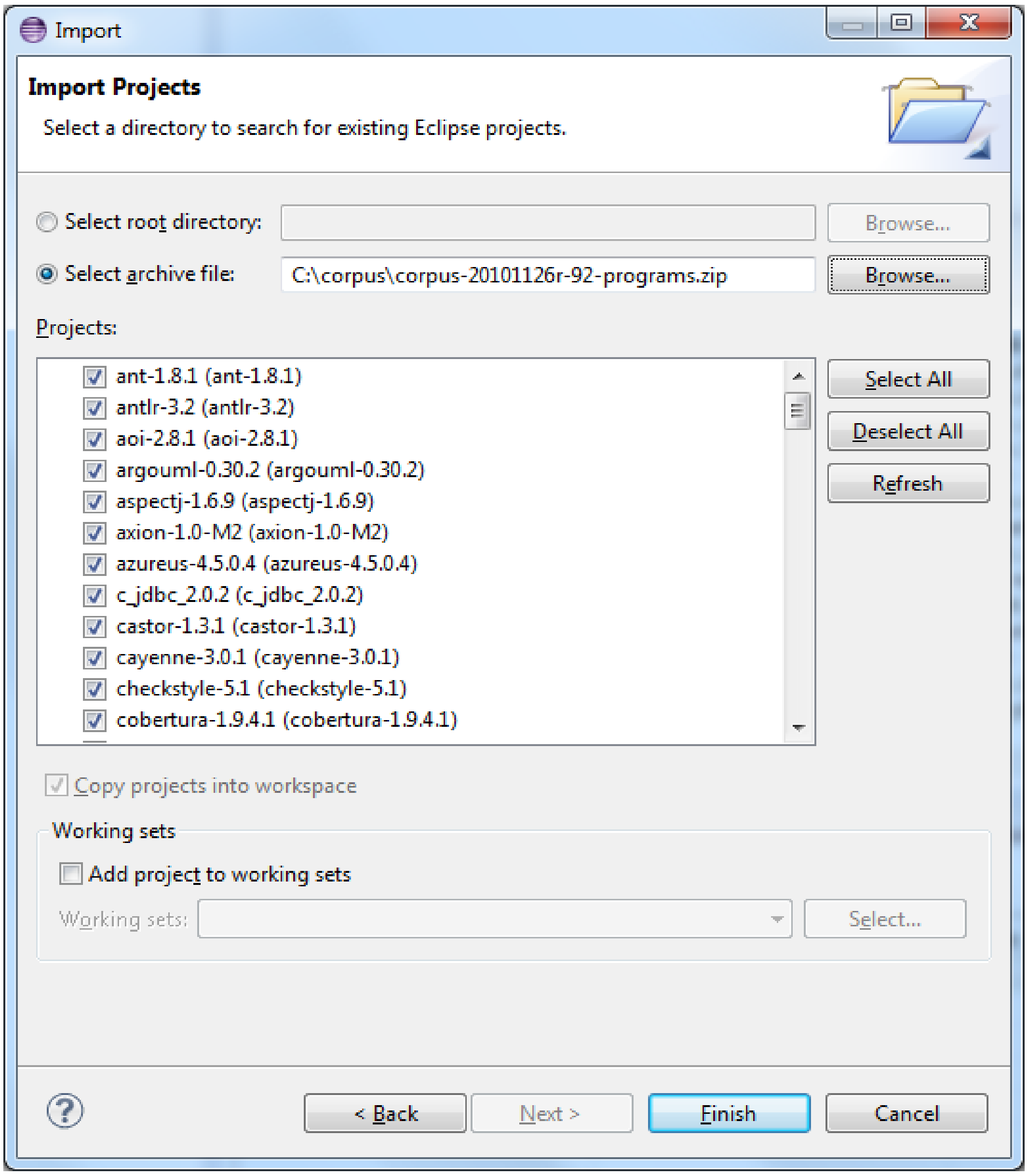


Figure B.4: Import Existing Projects



C:\corpus\corpus-20101126r-92-programs.zip

- [✓] ant-1.8.1 (ant-1.8.1)
- [✓] antlr-3.2 (antlr-3.2)
- [✓] aoi-2.8.1 (aoi-2.8.1)
- [✓] argouml-0.30.2 (argouml-0.30.2)
- [✓] aspectj-1.6.9 (aspectj-1.6.9)
- [✓] axion-1.0-M2 (axion-1.0-M2)
- [✓] azureus-4.5.0.4 (azureus-4.5.0.4)
- [✓] c_jdbc_2.0.2 (c_jdbc_2.0.2)
- [✓] castor-1.3.1 (castor-1.3.1)
- [✓] cayenne-3.0.1 (cayenne-3.0.1)
- [✓] checkstyle-5.1 (checkstyle-5.1)
- [✓] cobertura-1.9.4.1 (cobertura-1.9.4.1)

[✓] Copy projects into workspace

[ ] Add project to working sets

Working sets:

Figure B.5: Select Projects to Import

Appendix C

List of Acronyms

API	Application program interface
AST	Abstract syntax tree
AWD	Abstraction without decoupling
CI	Constructor invocation dependency
DEGINH	Degenerated inheritance
DG	Dependency graph
IN	Interface dependency
JDT	Java development tool
LTK	Eclipse refactoring language toolkit
MET	Method exception type dependency
MPT	Method parameter type dependency
MRT	Method return type dependency
QC	Qualitas Corpus
SC	Superclass dependency
SCC	Strongly connected components
SCD	Strong cyclic dependency
SMI	Static method invocation dependency
STK	Subtype knowledge
VD	Variable declaration dependency
WCD	Weak cyclic dependency