An Analytical Study of Metrics and Refactoring

by

Suchitra S. Iyer, B.Tech.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2009

An Analytical Study of Metrics and Refactoring

Approved by
Supervising Committee:

Dewayne E. Perry, Supervisor

Miryung Kim

To Lord Ganesha and Ayyappa

Acknowledgments

I am very grateful to my supervisor Dr. Dewayne E. Perry for guiding me throughout the course of my thesis. I attribute completing this thesis to his insightful suggestions and comments, patience and understanding. My sincere thanks to Dr. Miryung Kim for her constructive comments and suggestions on the thesis writeup.

I wish to acknowledge Dr. Danny Dig, University of Illinois at Urbana-Champaign, for sharing his ideas on refactoring detection in open source projects. Many thanks to Dr. Robert H. Flake for encouraging my pursuit of a graduate degree.

I wish to thank my parents, Satyabhama Suryanarayan and L. N. Surayanarayan for their boundless love and support. To them I dedicate this thesis. Many thanks to my loving husband Dr. Hari Mony for his words of encouragement and sharing his research experiences. Lastly, I wish to thank my sisters Sujata Seshadrinathan and Sunitha Raghuraman and their lovely families for all their support and good wishes.

SUCHITRA S. IYER

The University of Texas at Austin
May 2009

An Analytical Study of Metrics and Refactoring

Suchitra S. Iyer, M.S.E.
The University of Texas at Austin, 2009

Supervisor: Dewayne E. Perry

Object-oriented systems that undergo repeated modifications commonly endure a loss of quality and design decay. This problem is often remedied by applying refactorings. Refactoring is one of the most important and commonly used techniques to improve the quality of the code by eliminating redundancy and reducing complexity; frequently refactored code is believed to be easier to understand, maintain and test. Object-oriented metrics provide an easy means to extract useful and measurable information about the structure of a software system. Metrics have been used to identify refactoring opportunities, detect refactorings that have previously been applied and gauge quality improvements after the application of refactorings.

This thesis provides an in-depth analytical study of the relationship between metrics and refactorings. For this purpose we analyzed 136 versions of 4 different open source projects. We used RefactoringCrawler, an automatic refactoring

detection tool to identify refactorings and then analyzed various metrics to study whether metrics can be used to **(1)** reliably identify refactoring opportunities, **(2)** detect refactorings that were previously applied, and **(3)** estimate the impact of refactoring on software quality.

In conclusion, our study showed that metrics cannot be reliably used to either identify refactoring opportunities or detect refactorings. It is very difficult to use metrics to estimate the impact of refactoring, however studying the evolution of metrics at a system level indicates that refactoring does improve software quality and reduce complexity.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

All successful software is subject to change and dealing with change is one of the ”essential complexities” of developing software [Bro95]. As software grows, it undergoes continuous modifications often making its structure increasingly complex, hard to understand and hence, difficult to change. The laws postulated in [LPR+97] that suggest that software has a tendency to decay over time and becomes less understandable, maintainable and more complex have largely been proved true. In object-oriented systems, the quality characteristics of software can often be improved by applying certain behavior preserving restructurings called refactoring. Refactorings as defined by Fowler [Fow00] is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. He goes on to describe it as improving the design after it has been written. Thus, refactoring can be seen as a form of preventive maintenance. It makes it easier to add new code, improve the quality of existing code and gain a better understanding of the code by increasing its readability.

1.2 Motivation

In [Fow00], Fowler and Beck recommend a list of clues or symptoms that suggest refactoring opportunities. These symptoms or stinks are named *Bad smells* and in

his opinion, their detection should be achieved from *the programmer's intuition and experience*. It is upto the programmers to decide which refactoring to apply on the basis of the future anticipated changes for the system. Many past research efforts [SSL01, CLMM05] have focused on using object-oriented metrics to support these subjective perceptions of code smells to identify structures in need of refactoring. The aim is to use metrics to support decisions on where to apply which refactorings. Metrics have also been used to detect previously applied refactorings at each step of evolution [DDN01]. Closely related to the use of change metrics to detect refactorings is the idea of studying change metrics to estimate the impact of refactoring on software quality [SS07, HMKI08]. However, little academic work has been done in studying the effect of refactorings on object-oriented entities from the perspective of metrics. This thesis aims at analytically studying the relationship between metrics and refactoring. Such a study would help us understand if the use of metrics is indeed an effective way to identify refactoring opportunities, for detecting refactorings that may have occurred and studying the impact of refactorings on quality.

1.3 Research Problem

The goal of this thesis is to *analytically study the relationship between metrics and refactoring*. We try to achieve the research goal by attempting to answer the following questions:

1. Can metrics effectively identify bad smells in code to detect refactoring opportunities?

2. Can refactorings that have occurred be effectively detected via measuring change in code metrics?

3. Can the impact of refactoring be estimated by studying changes in metrics of the code after applying refactoring?

1.4 Research Method

To achieve our research goal, we analyze existing open source Java projects for refactorings. Rather than manually trying to detect refactorings by studying the code and the version history, we use the tool *RefactoringCrawler* [DCMJ06] to automatically detect refactorings in the code. We chose RefactoringCrawler because of its high accuracy, ease of use and availability as an eclipse plug-in. Once refactorings are detected, we generate various object oriented metrics for the two versions across which refactorings have been applied. We use the Eclipse metrics plug-in 1.3.6 [Sua] and VizzAnalyzer [VIZ], also available as an Eclipse plug-in to generate object oriented metrics. We chose the Eclipse metrics plug-in and the VizzAnalyzer since they calculate a comprehensive set of metrics related to quality as measured in terms of complexity, coupling and cohesion. In this thesis, we focus on analyzing metrics for three refactorings: Move methods, Pull up methods and Push down methods.

1.5 Thesis Outline

This chapter provides a high level understanding of the work under taken, in particular, the research goal and techniques have been explicitly stated. Chapter 2 describes the need to undertake refactoring activities and describes how and when to apply move method, pull up method and push down method refactorings. Chapter 3 describes various automatic refactoring detection techniques with particular focus on RefactoringCrawler. Chapter 4 defines and describes the object-oriented metrics and tools used in our analysis. Chapter 5 describes the experimental setup and presents the research questions we wish to answer through our study. Chapter 6 presents the analyses, observations and inferences for move method refactorings. Chapter 7 presents observations and results pertaining to pull up method refactorings. Chapter 8 deals with push down methods. Chapter 9 presents a system's view of various metrics with regard to versions that underwent refactoring. Chapter 10 includes a validity evaluation for our approach. We conclude our thesis with Chapter 11 that presents our results and ideas for future research.

Chapter 2

Refactoring and Decision Drivers

2.1 Refactoring: Introduction

Refactoring, introduced by Opdyke in his PhD dissertation [W.F92], refers to *the process of changing an object-oriented software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*. It is the object-oriented equivalent of *restructuring*, which is defined by Chikofsky and Cross [EJ90] as *the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior*. The focus is on altering internal structure of code to improve the design and hence maintainability, understandability and performance of code.

Fowler [Fow00] discusses refactoring as a secure code modification technique aimed at improving the program structure. The characteristics of ill-structured programs are what he calls *bad smells*. His work includes a comprehensive catalog of refactorings and presents a set of bad code smells associated with the cataloged refactorings. The sections below describe the need for refactoring, the various steps involved in applying a refactoring and a description of move method, pull up and push down method refactorings.

2.2 Why Refactor?

As a system ages, its design decays because maintenance efforts often concentrate more on bug fixing and on addition of new functionality than on improving its design [Bro95]. Bad design practices, often due to insufficient knowledge, lack of experience or time can make the structure brittle and can cause modularity to decrease. The code then becomes harder to understand and so future changes would still further contribute to the decay. The preferred technique for correcting ill-structured code is refactoring because the aim is to improve design without changing the system's observable behavior. Refactoring contributes constructively to several aspects of a software system such as readability, understandability, reuse, maintainability, changeability, and performance.

2.2.1 Readability and Understandability

Refactoring makes code more readable. It is important to be able to convey the intention of the code to others. It also makes the code easier to read for the original developer which is as much important since it is unrealistic to assume that a developer would remember all of the low-level design decisions. At the risk of sounding counter-intuitive, Fowler [Fow00] claims that the real advantage of refactoring is that it helps you develop software more quickly. Refactoring involves modification effort and a much larger testing effort. However, both bug fixing and modifying code to add new features is faster when the developer understands the system well, the design intentions are clear and the program is still well structured.

2.2.2 Reuse and Maintainability

Increases in quality and decreases in cost and implementation effort are benefits that can be derived from software reuse. In the closely related maintenance phase, the software system is further refined or augmented with new features. Both these activities need to be planned for. [BR89] suggests that all approaches that support reuse address the following four aspects: finding, understanding, modifying, com-

posing a component or set of components. Refactoring a program helps in all these aspects of software reuse [W.F92]. It makes program easy to understand and assists in finding suitable components for reuse. Some refactorings help modify a component making it easy to reuse and compose together for a system.

2.2.3 Changeability

High cohesion and low coupling between classes are considered attributes of good design. Systems with high levels of coupling face a number of issues including: a single change causing a ripple effect, modules becoming hard to reuse and test and difficulty in understanding a module in isolation. It is not always possible to get the design right in the first go. This can be due to a number of reasons including the development methodology followed, volatile nature of requirements etc. In Agile methodology, for instance, code is maintained and extended from iteration to iteration, and without continuous refactoring, this is hard to do. This is because un-refactored code tends to decay and the decay takes several forms: unhealthy dependencies between classes or packages, bad allocation of class responsibilities, too many responsibilities per method or class, duplicate code, and many other varieties of confusion and clutter that affect coupling and cohesion. When such decay is present, it is often hard to introduce a change in the form of a new feature or refine an existing one.

2.2.4 Performance

In cases where an application's design and/or implementation are responsible for poor performance, refactoring can be applied to specific locations in code to improve performance. For instance, the *Split Loop Refactoring* [Fow00] is often used for performance optimization in data intensive applications.

2.3 Refactoring Process

The survey paper by Mens and Tourwe [MT04] identifies six steps in the refactoring process:

1. Identify where the software should refactored.
2. Determine which refactorings should be applied to the portions identified.
3. Ensure that the applied refactoring is behavior.
4. Apply the refactoring.
5. Assess the effect of refactoring on quality characteristics.
6. Maintain the consistency between the refactored code and other software artifacts that use it.

Steps 1 and 2 are related in that identifying the need to refactor is driven by the type of refactoring one would apply. Many techniques have been proposed to automate steps 1 and 2 and most research is focused on identifying bad smells in the source code. Once identified, the information can be used to propose refactorings that can be used to reduce or remove the bad smell. For some basic refactorings, tool support is available for steps 3 and 4. Once a refactoring is applied, its effect typically is measured in terms of quality improvements. The goal of this thesis is to analyze the role object-oriented metrics plays in Steps 1, 2 and 5 for move methods, pull up methods and push down methods refactorings.

2.4 Refactoring Definitions

In our study of metrics and refactoring, we focus on some typical refactorings: move methods, pull up methods and push down methods. We briefly present these refactorings and their corresponding bad smells which can help identify parts of the system where they need to be applied.

2.4.1 Move Method

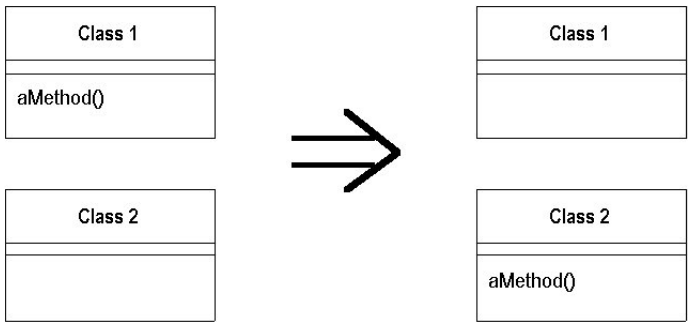

Figure 2.1: Move Method

A method m of class A is moved to another class B. The method *m* of class *A* can then be turned into a simple delegation or is deleted form class *A*. The bad smell motivating this refactoring is that a method uses or is used by more features of another class than the class in which it is defined. It is applicable when there are violations to the *put together what belongs together* principle. Move method refactoring is illustrated in Figure 2.1.

2.4.2 Pull Up Method

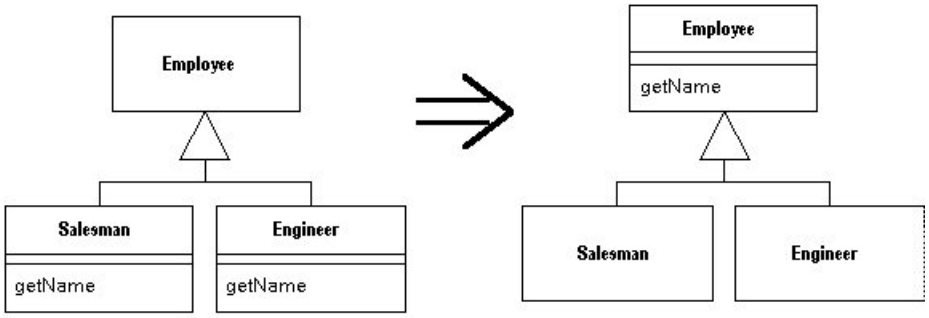

Figure 2.2: Pull Up Method

Pull up method refactoring involves *moving methods from subclasses with identical results to its superclass A*. Moving the shared functionality to the super-

class will leave the functionality of the subclasses unchanged, and give us less code duplication and better code readability. Pull up method refactoring is illustrated in Figure 2.2.

2.4.3 Push Down Method

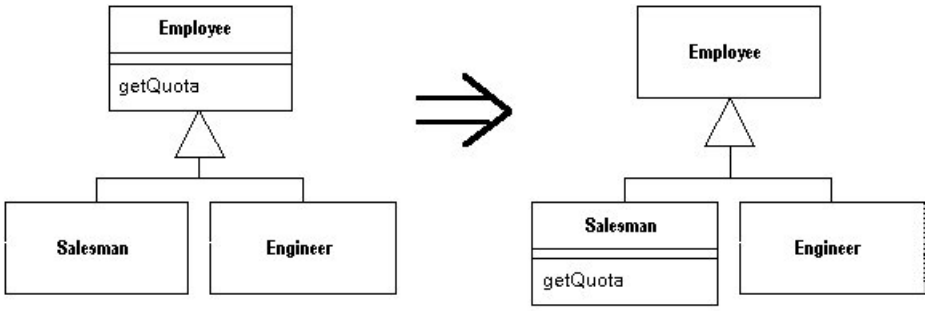

Figure 2.3: Push Down Method

Push down method is the opposite of pull up method. This is applicable when the behavior contained in the superclass is applicable to only some of its subclasses. This involves *moving a method m from a super class A to its subclasses*. Push down method refactoring is illustrated in Figure 2.3.

Chapter 3

Automatic Refactoring Detection

In order to evaluate how a refactoring decision affects software quality, we need to detect refactorings that have occurred between consequent versions of open-source software projects. Research in this area has focused on four key resources to collect such data: source code revision history, commit logs, logs obtained from refactoring tool usage and from programmers who have performed the refactorings.

If a project uses refactoring tools to perform refactorings, then tool usage logs can be used to find what kind of refactorings have happened in the past [HD05]. Such an approach will have very high accuracy but will miss all refactorings that were performed by hand without the use of the tool. Observing programmers and interviewing them can yield answers to where and what type of refactorings occurred during development. However, observing programmers during the course of development can prove to be very expensive. Also, if programmers are interviewed after the development process, we would be largely relying on the programmers ability to recall the exact portions of code where refactoring occurred.

Another way of identifying refactorings is by parsing the commit logs updated by programmers when a change is checked-in for words like *refactor*. By searching for more specific words like *moved method, rename, extract, push up, pull down etc.*, a researcher can find versions where particular refactorings have been applied. [RSG08] used 13 such keywords to find refactorings to conclude that an increase in refactorings is followed by a decrease in software defects. [HGH08]

used information obtained from commit logs to conclude that refactorings are more common when commits span many files. However, this technique assumes that programmers often include commit messages during check-in and also the correctness of the check-in comment. Hence, mining commit logs may not be accurate indicators of what refactorings have occurred.

An important technique used to detect what refactoring has occurred in the past uses source code version history. Object-Oriented change metrics has been used in [DDN01] to detect refactorings like *splitting and merging of classes*. They argue that discovering refactorings by textual comparisons would be extremely tedious and hence they use a set of heuristics to detect refactorings. UMLDiff tool [XS05] can help detect *move and rename refactorings*. However, a key assumption made by UMLDiff algorithm is that most entities remain same between two versions. Thus, when changes are huge (between major releases, infrequent check-ins by programmers), identifying potential moves and renames is not only time-consuming but also has lower precision. [FG06] computes edit operations by comparing abstract syntax tree representations of two programs and identify changes inside the same class. However, this algorithm cannot explicitly state the type of refactoring and misses higher level refactorings. [PD06] detect refactorings by using the information from code repositories and use clone detection to further refine the results. [RD03] use a clone finding tool (Duploc) to detect *move method* refactorings. [KNG07] automatically identifies API-level refactorings using a rule-based change inference approach. The focus is on recovering structural modifications to detect moves, renaming and other refactorings. RefactoringCrawler [DCMJ06] uses static and semantic analysis to detect refactorings. As can be noted from our discussion, all of the above detectors are limited by the types of refactorings they can detect. Most of the current tools are heuristics based, typically using a user supplied threshold value for heuristics. Hence, the user supplied parameters can affect the accuracy of detection results and using them will require some amount experimentation to determine appropriate threshold values. Another factor that needs to be addressed is regarding the two versions that are analyzed. Two adjacent versions analyzed may not have any important refactorings

and on the other hand, some fine-grained refactorings may be missed by analyzing non-adjacent versions.

Since the focus here is on finding how refactorings affect software quality, we need to use an approach that is highly accurate. Mining commit logs and interviewing or observing programmers to obtain past refactorings might not be accurate enough. Also, since we are analyzing open-source projects for refactorings, we cannot assume the use of refactoring tools to perform refactorings. Hence, we decided to use a source code analysis technique to detect refactorings. More specifically, we have used RefactoringCrawler to uncover past refactoring because of its high accuracy, ease of use and availability as an eclipse plug-in.

3.1 RefactoringCrawler

All the contents in this section have been borrowed from [DCMJ06]. The tool was put forth to address the issue of upgrading open-source applications when the components they use undergo change in the form of refactorings. It is used to detect component refactorings and then replay them on the applications that use them. It detects seven types of refactorings, including renamings, change of method signatures, moved methods etc., between two versions of Java components with about 85% accuracy.

The algorithm combines syntactic and semantic analysis to infer different refactorings. The syntactic analysis uses shingles encoding to identify similar entities and then the semantic analysis identifies the type of refactoring applied. The algorithm follows an order (top-down) in detecting rename refactorings (detects class rename, followed by method rename) and (bottom-up) for moved methods so that it can correctly infer cases where a single entity undergoes multiple refactorings. A user supplied similarity threshold value is used in syntactic analysis. A very high similarity threshold value can result in the tool missing relevant refactorings because it would then look for near perfect matching entities. The results obtained from syntactic analysis represent candidate entities. These are further analyzed using semantic analysis to determine whether they represent a refactoring.

The syntactic analysis parses files in two versions into lightweight abstract syntax trees. This parsing stops at the declaration of methods and attributes. Each version is represented as a graph where a node represents a source level entity. Nodes are arranged hierarchically. The shingles are *fingerprints* for strings with the following property: if a string changes slightly, then its shingles changes slightly. The result of the syntactic analysis step is a set of pairs of entities that have similar shingles encoding in two versions. For the similar pair to be considered a candidate refactoring, its normalized similarity value must be above a user specified threshold value. The semantic analysis is based on reference graphs that represent references among source-level entities. It compares the similarity of references to a user-specified threshold value.

To find moved method candidates, a second syntactic check is performed to ensure that the parent classes of the two methods are different. Without this check, a moved method would incorrectly classify all methods of a renamed class as moved methods. Another semantic check requires that the declaration classes of methods not be related by inheritance, otherwise it would be classified incorrectly as a moved method instead of a push down or a pull up method. The last check requires that all references to the target class be removed in the second version and that all calls to methods from the initial class be replaced with sending a message to an instance of the initial class. A pull up method pulls up the declaration of a method from a subclass into the superclass such that the method can be reused by other subclasses. A push down method pushes down the declaration of a method from a superclass into a subclass that uses the method because the method is no longer reused by other subclasses. For push up and pull down method it checks to ensure the original class is a descendant and an ancestor respectively of the target-class.

3.2 Strengths

1. High precision and recall
   Evaluation results for RefactoringCrawler have shown that both precision and recall over 85%. Compared to approaches that use only syntactic analysis and

produce a large number of false positives, RefactoringCrawler requires little human intervention to validate the refactorings.

2. Robustness
   RefactoringCrawler can deal with noise associated with preserving backward compatibility and with multiple refactorings happening to same or related entities.

3. Scalability
   The algorithm combines a relatively inexpensive syntactic analysis with the expensive semantic analysis.

3.3 Limitations

1. Poor support for interfaces and fields
   A key issue with the algorithm is that since it relies on code similarity to find whether a program element in one version matches with another, it has problems with elements that do not have a body.

2. Requires experimentation
   Selecting appropriate values for threshold values is important. Too high a value can miss some refactorings whereas too low a value can produce a large number of false positives.

3.4 Input & Output

Input:

1. Versions of projects to be compared.

2. Threshold values for various similarity parameters.

Output: The output of RefactoringCrawler details the pairs of refactorings detected. This output can be exported into an XML file.

Since the correctness of our analysis results depends largely on the accuracy of detected pairs of refactorings, we manually confirmed the results by going through the source code of the program versions. Some false positives were eliminated in this process. We also experimented with various values for the input similarity threshold values starting with high threshold values and realized that our conclusions tallied with those of the authors of this tool. As recommended, we used values between 0.5 to 0.7.

Chapter 4

Object-Oriented Metrics

Object-Oriented programming paradigm emerged in the 1960s and has since become increasingly popular in industrial software development environments. It was developed as a methodology in part to solve some of the issues with existing paradigms. The focus in object-oriented systems is on data rather than processes, with programs in object-oriented languages composed of modules called objects each comprising of functions to manipulate its own data structure. Software metrics have long been proved to reflect software quality and thus, used widely in software maintainability evaluation methods [BJM76]. As the use object-oriented languages became more widespread, so did the need to ensure the quality of software written in object-oriented languages. For object-oriented systems, many metrics suites have been proposed. These include suites proposed by Henry and Kafura [SD81, SD84], Chidamber and Kemerer [SC91, SC94], Li and Henry [WS93], Lorenz and Kidd [MJ94], Henderson-Sellers [HS96] and Briand et al. [LPW97, LSV99]. Other popular metrics suites include, McCabe's Cyclomatic Complexity [T.J76, TC89, TA94] and Halstead's Complexity measures [M.H77].

In this thesis, we studied various object-oriented metrics for the following purposes:

- We analyzed pre-refactored version of source code to evaluate the correlation

between bad smells in code and object oriented metrics used to identify them as in [SSL01, CLMM05].

- We analyzed object-oriented metrics for pre- and post-refactored versions of source code to identify if indeed change metrics between the two versions can be used to detect refactorings that may have occurred as in [DDN01]

- We analyzed object-oriented metrics for pre- and post-refactored various of source code to measure the improvement in quality after refactoring and analyze the impact these refactorings have on quality as in [HMKI08]

4.1 Object-Oriented Metric Tools

A number of commercial and non-commercial metric tools support the collection and analysis of software metrics. For the purpose of our thesis, a tool that would calculate a defined set of metrics, related to quality as measured in terms of complexity, coupling and cohesion was required. For ease of use, tools available as Eclipse plug-ins were preferred. In order to automate the process of extracting refactorings and comparing metrics before and after refactoring, a tool that had the capability of presenting the output in a structured format such as XML was desired. Eclipse Metrics plug-in 1.3.6 [Sua] by Frank Sauer is an open source metrics calculation and dependency analyzer plug-in for the Eclipse IDE. This plug-in computes most required metrics and the output can be exported into an XML file. We have used this plug-in to analyze metrics before and after refactoring for all the Java based open source projects under consideration. The plug-in uses [HS96, Mar03] as its primary source for the definitions of complexity, coupling and cohesion metrics. However, this plug-in does not compute two metrics that were important for our analysis that gave a measure of coupling and complexity for a Java class. To overcome this, another tool VizzAnalyzer was used [VIZ]. VizzAnalyzer is a commercial tool that is also available as an Eclipse plug-in. The results can be copied or dumped into comma separated files (CSV).

4.2 Overview of Metrics Analyzed

Some of the metrics we investigated for the open-source projects are listed below. It can be argued that these metrics may be more representative of program size than of quality. However, previous research efforts have successfully demonstrated them as indicators of software quality as in [VM96].

1. Number of Attributes (NOF)
   NOF is a class level metric that gives a count of number member variables in a class. It indicates the amount of data the class must maintain in order to carry out its responsibilities.

2. Number of Methods (NOM)
   NOM is a class level metric that gives a count of number of methods in a class. It indicates the total level of functionality implemented by a class.It provides a view of class size and how complex the class might be to use.

3. Number of Static Attributes (NSF)
   NSF is a class level metric that gives a count of static member variables of a class. A high value potentially restricts reuse.

4. Number of Static Methods (NSM)
   NSM is a class level metric that gives a count of static methods in a class. Since, static methods are not instantiated, a high value of this metrics indicates restricted reuse.

5. Number of Children (NSC)
   NSC is a class level metric. It is calculated as total number of direct subclasses of a class. A class implementing an interface counts as a direct child of that interface. High values of NSC can indicate improper abstraction of the parent class. Also, such parent classes are hard to modify since a change potentially affects all its children. It can be an indication that it may be necessary to group related classes together and introduce another level of inheritance.

6. Depth of Inheritance Tree (DIT)

   DIT is a class level metric. It gives the distance of a class from class Object in Java. The greater the distance from Object class, greater are the number of methods it is likely to inherit, making it more complex as far as class behavior is concerned. High values of DIT also indicate greater design complexity, since more methods and classes are involved.

7. Weighted Methods per Class (WMC)

   WMC is a class level metric. It is obtained by computing the sum of the McCabe Cyclomatic Complexity for all methods in a class. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Given E to be the number of edges of the graph, N to be the number of nodes of the graph and P to be the number of connected components (for a single program/method, $P == 1$), the cyclomatic complexity [T.J76] M is then defined as:

   Definition 4.1. $M = E - N + 2P$

   The WMC value computed using the number of methods and their complexities, is a predictor of how much time and effort is required to develop and maintain the class. Classes with large numbers of methods and hence, a high WMC value, are likely to be more application specific, limiting the possibility of reuse.

8. Coupling Between Object classes (CBO)

   CBO is class level metric that provides the number of classes to which a given class is coupled. A class is coupled to another of if it uses its member functions and/or instance variables. High levels of coupling is detrimental to modular design and prevents reuse.

9. Response for a Class (RFC)

   The response set of a class is a set of methods that can potentially be exe-

cuted in response to a message received by an object of that class. RFC is simply the number of methods in the set. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Increasing RFC decreases analyzability, understandability, changeability and testability.

10. Lack of Cohesion of Methods (LCOM)
    LCOM is a class level metric that computes cohesiveness of a class. We have used Henderson-Sellers definition of LCOM in our study [HS96]. Consider a set of methods $\{M_i\}$ $(i = 1, \ldots, m)$ accessing a set of attributes $\{A_j\}$ $(j = 1, \ldots, a)$. Let the attributes accessed by each method, M_i, be written as $\alpha(M_i)$ and the number of methods which access each datum be $\mu(A_j)$. LCOM is defined as follows:

    Definition 4.2. LCOM = $\frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right)-m}{1-m}$

    High levels of cohesion implies that the features of a class are extensively used. LCOM measures the correlation between the methods and the local variables of a class. The value varies from 0 to 1, with the value zero representing perfect cohesion and the value one presenting extreme lack of cohesion. A high LCOM value indicates decreased encapsulation and increased complexity while a low value implies high cohesion and good design. Lack of cohesion implies that the classes should probably be split into two or more subclasses.

11. Afferent Coupling (CA)
    CA is a package level metric that computes the number of classes outside a package that depend on classes inside the package. It is an indication of the package's responsibility.

12. Efferent Coupling (CE)

CE is a package level metric that computes the number of classes inside a package that depend on classes outside a package. It is an indication of the package's independence.

13. Instability (RMI)
    RMI is a package level metric. Instability is computed as:

    Definition 4.3. RMI = $\frac{CE}{(CA+CE)}$

    It is an indication of a package's resilience to change. RMI has a range [0,1] where, 0 indicates a maximally stable package and 1 indicates a maximally unstable package.

Chapter 5

Metrics and Refactoring

In this chapter, we analyze various open source projects to study the relationship between software metrics and refactoring. The relationship between metrics and refactoring has been widely researched. This research can be classified into three categories:

1. **Metrics as hints for identifying refactoring opportunities**
2. **Metrics for refactoring detection**
3. **Metrics for studying refactoring impact**

Metrics as hints for identifying refactoring opportunities: Refactoring is considered the key to increasing software quality during the whole software life-cycle. Though many different kinds of refactorings have been cataloged [Fow00], one of the main problems in applying refactoring on large systems is the question of where to apply which refactoring. This question becomes even more difficult since refactorings are typically applied based on *human intuition*. Traditionally the decision on where to refactor is based on *code smells*. A bad code smell is a structure that needs to be removed from the source code by refactoring to improve the maintainability and testability of the software. Examples of bad code smells include long classes, long methods, large number of public methods etc.

Software metrics provide an easy means to extract useful and measurable information about the structure of a software system. Use of appropriate software metrics has been proposed as an objective measure of identifying bad code smells in the design [SSL01, CLMM05]. Even though Fowler explicitly mentions [Fow00] that no metrics can rival informed human intuition, previous research [SSL01] has shown metrics can help identify particular anomalies in source code associated with certain refactorings.

Metrics for refactoring detection: Reverse engineering is an ongoing process in the development of any successful software system as it evolves in response to changing requirements. Given that refactoring is integral to improving software quality, it would be very useful to get a better understanding the previous refactoring decisions. Essentially, one would like to find out which parts of the design has undergone refactoring, what type of refactoring was applied and why did the designers chose to apply refactorings in those particular parts of the design.

A variety of ways to detecting refactorings have been described in Chapter 3. As described in Chapter 3, changes in object-oriented code metrics has also been proposed as a way to detect refactoring [DDN01]. A set of heuristics were proposed in [DDN01] to detect refactorings by applying lightweight object-oriented metrics to successive versions of a software system. Each heuristic is defined as a combination of change metrics which reveal refactorings of a certain kind. The heuristics aim at focusing attention on the relevant parts of the software system to detect refactorings. One heuristic may occasionally miss refactorings or misclassify them, but such mistakes are typically corrected by one of the other heuristics.

Metrics for studying refactoring impact: Closely related to the use of change metrics to detect refactorings is the idea of studying change metrics to estimate the impact of refactoring on software quality [SS07, HMKI08]. Software metrics have been proved to reflect software quality. In [SS07], the authors analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected

by this process, in order to evaluate whether refactoring is effectively used as a means to improve software quality within the open source community.

It is typically difficult to perform appropriate refactorings since the impact of refactoring should justify the cost. In [HMKI08], the authors try to estimate the effect of refactoring by analyzing the metrics associated with structural changes of the source code, in particular,

- How does coupling between classes change?
- How does cohesion of each class change?
- How does the inheritance relationships between classes change?

In [HMKI08], the authors develop a software tool which outputs quantitative result of the effect estimation from the viewpoint of the above three questions.

We aim to answer the following three questions based on the analysis of open source projects:

1. Can metrics enable us to identify good refactoring opportunities ?
2. Can change metrics be reliably used to identify refactorings?
3. Can metrics be used to evaluate impact of refactorings?

To obtain the answer to the above questions, we propose to do the following. We first analyze existing open source projects to detect refactorings using an automatic refactoring detection tool that does not use metrics to detect refactoring. Next, we will generate software metrics for each version of the open source project analyzed. For each detected refactoring, we will study the associated metrics and try to answer the questions described above. Our study is qualitatively different from previous research in each of the above three areas.

- Though there has been a previous research effort [SSL01, CLMM05] to correlate bad smells with applied refactorings, refactorings detected by automatic refactoring detection techniques have not been used for this analysis. The

previous analysis was restricted to refactorings detected using change logs. Our analysis is more comprehensive in that designers might have forgotten to log certain refactorings which can be captured by the automatic refactoring detection program.

- Though there has been study on validating refactorings identified using change metrics [DDN01], there has not been much comparison between automatic refactoring detection techniques that do not use metrics and refactoring detection techniques that use metrics.

- Our refactoring impact analysis has several features which makes it qualitatively different from previous research. In [SS07], when studying the impact of refactoring on software quality, the authors used version logs to identify refactorings as opposed to our approach where we use an automatic detection tool. Also, they did not try to correlate each kind of refactoring to a specific trend in the change of various metrics. In [HMKI08], the authors do attempt to estimate the impact of each type of refactoring using metrics. However, there are very few results to draw any conclusion on the estimation of the impact of refactoring using metrics. The authors mention pull up method refactoring, but only analyze move method refactoring. Also, only a single move method was considered in the paper.

This chapter is organized as follows. We describe our experimental setup and give an overview of various open source projects analyzed in Section 5.1. The various issues we encountered when mining for refactorings in the various open source projects are described in Section 5.2.

The experimental results from move method refactorings identified from the analysis along with our observations and inferences is described in Chapter 6. Experimental results, observations and inferences for pulled up method refactorings and pushed down method refactorings are described in Chapter 7 and Chapter 8 respectively.

OSS Project	CAROL	DNSJAVA	STRUTS	TOMCAT
Lines of Code	7159	14334	12850	155832
No: Classes	129	149	133	1223
No: Packages	35	5	15	102
Last Release	Carol 2.2.10	dns 2.0.6	Struts 2.1.6	Tomcat 6.0.18
Duration	34 months	104 months	89 months	96 months
No: Versions Analyzed	70	43	11	12
URL	carol.objectweb.org	dnsjava.org	struts.apache.org	tomcat.apache.org

Table 5.1: Open Source Projects Analyzed

5.1 Experimental Setup

The focus of our refactoring analysis is open-source subject programs written in Java. We selected Apache Struts, DNSJAVA, CAROL and Tomcat that satisfy this condition and have their source code repositories available. We chose the two Apache software foundation projects (Struts and Tomcat) for our study due to the fact that they are well-documented, mature and popular open source projects. Tomcat also fulfilled our requirement for a fairly big-sized open source project. CAROL and DNSJAVA were chosen due to the author's familiarity with the projects and due to the fact that they are small enough to enable manual analysis.

1. Apache Struts is an open-source framework for creating Java web applications. It has evolved considerably between its first release made in July 2001 and its last Struts 2.1.6 in Jan 2009.

2. DNSJAVA is an implementation of Domain Name Service in Java and has evolved over 9 years between its first release in September 1998 and its last dnsjava 2.0.6 in January 2009.

3. CAROL is a library that allows clients to use different RMI implementations and has evolved over 34 months between August 2002 and May 2005.

4. Apache Tomcat is a servlet container for the implementation of Java servlets

and Java Server Pages (JSP) technologies. The first release was in 1999 tomcat 3.0 and the last one Tomcat 6.0.18 in July 2008.

We restricted our analysis to released versions of the above open source projects. The released versions were identified either using tags specified in their SVN repositories (CAROL and Apache Struts) or through source downloads available in their respective websites (DNSJAVA and Apache Tomcat). We analyzed 70 versions of CAROL, 43 versions of DNSJAVA, 11 versions of Apache Struts and 12 versions of Apache Tomcat. The 136 versions of the above open source projects were analyzed for refactorings using RefactoringCrawler [DCMJ06].

As described in Chapter 3, RefactoringCrawler is an eclipse-plug-in that detects refactorings between two versions of Java components. It has the ability to detect seven types of refactorings, including renamings, change of method signatures, moved methods, etc. RefactoringCrawler was run between adjacent versions of each of the above four open source projects. The tool was able to identify five different types of refactorings. These include:

1. Moved Methods
2. Pulled Up Methods
3. Renamed Classes
4. Renamed Methods
5. Changed Method Signatures

The results obtained using RefactoringCrawler were further analyzed and refined by excluding false positives. The analysis was done through manual inspection of the source code to identify false positives. A detailed analysis of false positives identified among RefactoringCrawler results is presented in Section 5.2.

5.2 RefactoringCrawler: False Positives

In this section, we detail the false positives encountered when using RefactoringCrawler to identify refactorings in various open source projects. Each refactoring detected by RefactoringCrawler was checked through manual inspection of the source code. The false positives were identified during this manual inspection of the source code.

Of the various refactorings identified by RefactoringCrawler, we only encountered false positives in move method refactorings. In addition to the 10 versions where RefactoringCrawler correctly detected move method refactorings, another five versions were incorrectly identified as having undergone move method refactoring. We marked a move method refactoring as being false positive if the source class (the class from which the method is moved from) does not exist in the new version (the version after refactoring has been applied). The only exception we have made is for a case where the destination class existed in both old (the version on which refactoring is applied) and new versions. The false positives we encountered can be classified into four main categories:

1. Renaming of classes: Classes were just renamed, but RefactoringCrawler thought that methods were moved into the renamed class.

2. Renaming of packages: Package which contained the methods was renamed causing incorrect detection of move method refactoring.

3. Removal of old package and moving classes from old package to different existing packages

4. Moving of classes from one package to another

The last two categories could possibly considered a move method since the methods were moved between packages, however we chose to classify them as false positive since the methods were not moved out of the class to which they belonged.

Chapter 6

Move Method Refactoring

In this chapter, we will analyze the results obtained using the experimental setup described in Chapter 5 to study the relationship between metrics and move method refactorings.

OSS Version	NOC	NOC with Move Methods	NOP	NOP with Move Methods
CAROL 1.2	49	2	12	1
CAROL 1.5.2	127	2	18	1
CAROL 1.8.9.4	161	1	24	1
CAROL 2.0.8	158	1	23	1
CAROL 2.2.9	108	3	23	3
DNSJAVA 1.6.6	146	1	4	1
STRUTS 1.1	332	2	41	1
STRUTS 1.2.7	345	1	40	1
TOMCAT 6.0.4	1163	2	100	2
TOMCAT 6.0.10	1189	1	100	1

Table 6.1: Move Method Refactoring

RefactoringCrawler detected move method refactorings in 10 versions out of the total 136 versions analyzed. Table 6.1 provides details of the versions where move method refactorings were detected. The first column describes the version, the second column describes number of classes (NOC), the third column describes

number of classes with move method refactorings and the fourth and fifth column provides similar information for the number of packages (NOP).

This chapter is organized as follows. In Section 6.1, we will describe some high level observations on the move method refactorings detected by RefactoringCrawler. The detailed experimental data on metrics obtained on the versions which underwent refactoring is described in Section 6.2. We detail our observations on the experimental results and our inferences based on the observations in Sections 6.3, 6.4, and 6.5. Section 6.3 tries to see whether metrics can enable us identify good move method refactoring opportunities. Section 6.4 analyzes whether change metrics can be reliably used to identify move method refactorings. Section 6.5 answers the question on whether metrics can be used to evaluate the impact of move method refactorings.

6.1 Observations:

- The move method refactorings detected by RefactoringCrawler can be broadly classified into three categories:

    1. The source and destination classes for the move method refactoring are members of the **same** package.
    2. The source and destination classes for the move method refactoring are members of **different** existing packages.
    3. The source and destination classes are members of different packages and the destination package did not exist in the version where refactoring was applied.

- Other then **DNSJAVA 1.6.6**, the destination class did not exist in the version where refactoring was applied. The methods were moved into a newly created class.

- In **DNSJAVA 1.6.6**, the source class does not exist in the new version after

refactoring. This can be classified as a special case of move method refactoring.

- RefactoringCrawler detected a large number of move method refactorings which were false positive in nature. The details are described in Section 5.2.
- The number of lines of code of methods which were moved varied from **1** to **56**. The average method lines of code for move method refactorings was **12.40** while the median was **6.75**.
- The ratio of number of methods moved from a class with respect to total number of methods in the class varied from **60 %** to **5.4 %**. The highest number of methods moved from a class was **22** while the lowest was **1**.

6.2 Experimental Data

OSS Version	Old (From)	Old (To)	Old (AVG)	New (From)	New (To)	New (AVG)
CAROL 1.2	0.667	-	0.748	1	1	0.792
CAROL 1.5.2	0.583	0.583	0.824	0.615	0.615	0.827
CAROL 1.8.9.4	0.9	0.667	0.726	0.909	0.6	0.677
CAROL 2.0.8	0.4	0.4	0.613	0.429	0.429	0.588
CAROL 2.2.9 (1)	0.412	-	0.571	1	0.278	0.601
CAROL 2.2.9 (2)	0.471	-	0.571	0.2	0.5	0.601
CAROL 2.2.9 (3)	1	-	0.571	1	0.6	0.601
DNSJAVA 1.6.6	0.878	0.878	0.678	0.556	0.556	0.731
STRUTS 1.1	0.091	-	0.68	0.101	0.019	0.695
STRUTS 1.2.7	0.357	-	0.69	0.2	1	0.498
TOMCAT 6.0.4 (1)	0.714	0.714	0.442	0.714	0.714	0.442
TOMCAT 6.0.4 (2)	0.368	0.368	0.443	0.368	0.368	0.442
TOMCAT 6.0.10	0.5	0.491	0.445	0.5	0.5	0.446

Table 6.2: Move Method Refactoring – RMI Data

OSS Version	Old (From)	Old (To)	Old (AVG)	New (From)	New (To)	New (AVG)
CAROL 1.2 (1)	0	-	0.119	0	0	0.105
CAROL 1.2 (2)	0	-	0.119	0	0	0.105
CAROL 1.5.2 (1)	0	-	0.16	0.57	0	0.165
CAROL 1.5.2 (2)	0	-	0.16	0.21	0	0.165
CAROL 1.8.9.4	0	-	0.18	0	0	0.178
CAROL 2.0.8	0.87	-	0.15	0.87	0	0.15
CAROL 2.2.9 (1)	0	-	0.11	0	0	0.101
CAROL 2.2.9 (2)	0	-	0.11	0	0	0.101
CAROL 2.2.9 (3)	0	-	0.11	0	0	0.101
DNSJAVA 1.6.6	0.61	0.83	0.19	-	0.78	0.18
STRUTS 1.1 (1)	0	-	0.29	0	0	0.28
STRUTS 1.1 (2)	0	-	0.29	0	0	0.28
STRUTS 1.2.7	0.82	-	0.28	0	0.78	0.24
TOMCAT 6.0.4 (1)	0.87	-	0.32	0	0.87	0.32
TOMCAT 6.0.4 (2)	0.72	-	0.32	0	0.72	0.32
TOMCAT 6.0.10	0.93	-	0.32	0	0.91	0.32

Table 6.3: Move Method Refactoring – LCOM Data

In this section, we detail the experimental data. In a move method refactoring, the class/package which underwent refactoring is referred to as **old** and the class/package into which methods were moved as **new**. For each move method refactoring detected by RefactoringCrawler, we analyzed the metrics on the old and new versions. The following four metrics were analyzed:

- Instability (RMI)
- Lack of Cohesion of Methods (LCOM)
- Coupling between Object Classes (CBO)
- Response for a Class (RFC)

Table 6.2 details the data obtained for RMI metric. Data is provided for each of the 13 packages which underwent move method refactoring. Column 1

OSS Version	Old (From)	Old (To)	Old (AVG)	New (From)	New (To)	New (AVG)
CAROL 1.2 (1)	1	-	2.09	1	2	2
CAROL 1.2 (2)	1	-	2.09	1	2	2
CAROL 1.5.2 (1)	8	-	2.16	9	9	2.25
CAROL 1.5.2 (2)	5	-	2.16	6	9	2.25
CAROL 1.8.9.4	5	-	2.32	5	4	2.31
CAROL 2.0.8	5	-	2.86	6	1	2.86
CAROL 2.2.9 (1)	8	-	3.02	1	8	2.68
CAROL 2.2.9 (2)	4	-	3.02	1	4	2.68
CAROL 2.2.9 (3)	1	-	3.02	1	1	2.68
DNSJAVA 1.6.6	3	25	5.58	-	22	5.48
STRUTS 1.1 (1)	2	-	3.83	2	11	3.67
STRUTS 1.1 (2)	22	-	3.83	18	11	3.67
STRUTS 1.2.7	10	-	3.64	9	3	3.90
TOMCAT 6.0.4 (1)	14	-	5.23	10	14	5.23
TOMCAT 6.0.4 (2)	3	-	5.23	1	3	5.23
TOMCAT 6.0.10	20	-	5.21	25	7	5.20

Table 6.4: Move Method Refactoring – CBO Data

lists the open source project version in which refactoring was detected. Column 2 provides the RMI value for the package which underwent refactoring (referred to as **FROM** in the Table) in the old version. Column 3 provides the RMI value for the package into which methods were moved to (referred to as **TO** in the Table) in the old version. Column 4 provides the average RMI value across all packages in the old version. Column 5 illustrates the RMI value for the **FROM** package in the old version and Column 6 illustrates the RMI value for the **TO** package in the new version. Finally, Column 7 illustrates the average RMI value across all packages in the new version.

Table 6.2 provides more details on the various move method categories discussed in Section 6.1. **CAROL 1.2, 2.2.9** and **STRUTS 1.1, 1.2.7** illustrate move method refactorings from one package to a newly created package. **CAROL 1.5.2, 2.0.8**, **DNSJAVA 1.6.6**, and **TOMCAT 6.0.4** illustrate move method refactorings within the same package. **CAROL 1.8.9.4** and **TOMCAT 6.0.10** illustrate move

OSS Version	Old (From)	Old (To)	Old (AVG)	New (From)	New (To)	New (AVG)
CAROL 1.2 (1)	2	-	7.68	1	4	0.105
CAROL 1.2 (2)	2	-	7.68	1	4	0.105
CAROL 1.5.2 (1)	18	-	7.63	19	22	7.79
CAROL 1.5.2 (2)	13	-	7.63	20	22	7.79
CAROL 1.8.9.4	35	-	8.99	7	30	8.41
CAROL 2.0.8	23	-	7.64	23	2	7.49
CAROL 2.2.9 (1)	19	-	7.67	11	19	7.18
CAROL 2.2.9 (2)	9	-	7.67	5	14	7.18
CAROL 2.2.9 (3)	5	-	7.67	4	9	7.18
DNSJAVA 1.6.6	16	63	12.5	-	66	12.52
STRUTS 1.1 (1)	5	-	12	7	48	12.04
STRUTS 1.1 (2)	83	-	12	102	48	12.04
STRUTS 1.2.7	33	-	12.07	28	10	12.21
TOMCAT 6.0.4 (1)	44	-	17.67	1	41	17.67
TOMCAT 6.0.4 (2)	11	-	17.67	1	11	17.67
TOMCAT 6.0.10	82	-	17.33	24	31	17.32

Table 6.5: Move Method Refactoring – RFC Data

method refactorings from one package to a different existing package.

Tables 6.3, 6.4, and 6.5 details the data obtained for class level metrics LCOM, CBO and RFC respectively. Data is provided for each of the 16 classes which underwent move method refactoring. In each of the Tables, column 1 lists the open source project version in which refactoring was detected. Column 2 provides the metric value for the class which underwent refactoring (referred to as **FROM** in the Table) in the old version. Column 3 provides the metric value for the class into which methods were moved to (referred to as **TO** in the Table) in the old version. Column 4 provides the average metric value across all classes in the old version. Column 5 illustrates the metric value for the **FROM** class in the old version and Column 6 illustrates the RMI value for the **TO** class in the new version. Finally, Column 7 illustrates the average metrics value across all classes in the new version.

6.3 Bad Smells to Identify Refactoring Opportunities

Move method refactoring is applied if a method is, or will be, using or used by more features of another class than the class on which it is defined. Bad smells that indicate a move method refactoring opportunity include *Shotgun Surgery, Feature Envy, Message Chains, Inappropriate Intimacy, Data Class etc.* [Fow00]. The *Shotgun Surgery* bad smell indicates a situation when a change in one class requires cascading changes in a number of other classes. The *Feature Envy, Message Chains, Inappropriate Intimacy* are bad smells that indicate high coupling. They are a result of deviation from *putting together what belongs together*. The *Feature Envy* smell means a case where one method is too interested in other classes, and the *Inappropriate Intimacy* smell means that two classes are coupled tightly to each other. The *Message Chains* smell implies a scenario where client is coupled to the navigation structure. The issue here is that the overhead of calling procedures can become significant which this indicates inefficient design. The *Data Class* bad smell indicates a situation where the class is not doing much and behavior needs to be moved into the class. The following object-oriented metrics are typically used to figure out bad smells to identify move method refactoring opportunities.

- **Instability (RMI)** RMI is a measure of instability of a package. A high value of RMI indicates that number of classes inside the package that depend on classes outside the package is much higher when compared to the number of classes outside the package that depend on classes inside the package. A high value of RMI could thus be a bad smell indicating that methods should be moved from classes inside the package to classes outside the package.

- **Lack of Cohesion of Methods (LCOM)** LCOM is a measure of cohesiveness of a class. A value close to **1** indicates lack of cohesion and could thus be a bad smell indicating that methods should be moved from the class to another class.

- **Coupling Between Object Classes (CBO)** CBO is a count of the number of other classes to which it is coupled. Inter-class couples should be minimized

as much as possible, because of re-usability, maintenance and modularity. High levels of coupling is detrimental to modular design and could possibly reduced through move method refactoring.

- **Response for a Class (RFC)** The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is a measure of potential communication between the class and other classes and one way to decrease RFC is through a move method refactoring.

6.3.1 Observations and Inferences:

1. A high value of RMI can only be considered a bad smell if the move method refactoring involves a move from one package to a different package. Of the 13 packages that underwent move method refactorings, only 2 packages (packages in **CAROL 1.8.9.4** and **TOMCAT 6.0.10**) had their methods moved to a different package (the rest were moves to another class in the same package or to a newly created package).

   We compared the RMI of each package with the average RMI computed across all packages in the particular version. RMI for the package in **TOMCAT 6.0.10** was higher than the average, but the difference was much less than the standard deviation. In fact, out of 100 packages, 19 packages had the highest possible RMI value of 1. RMI for the package in **CAROL 1.8.9.4** was higher than average and the difference was close to the standard deviation. However, there were 9 packages which had a higher RMI than the package which underwent refactoring and all of them had the highest possible RMI value of 1.

   To enable further analysis of RMI in **CAROL 1.8.9.4** and **TOMCAT 6.0.10**, we built boxplots for RMI values of all packages in the respective versions. The boxplots obtained are illustrated in Figures 6.1 and 6.2 respectively. Interestingly, there are no outliers in both the boxplots. This indicates that RMI

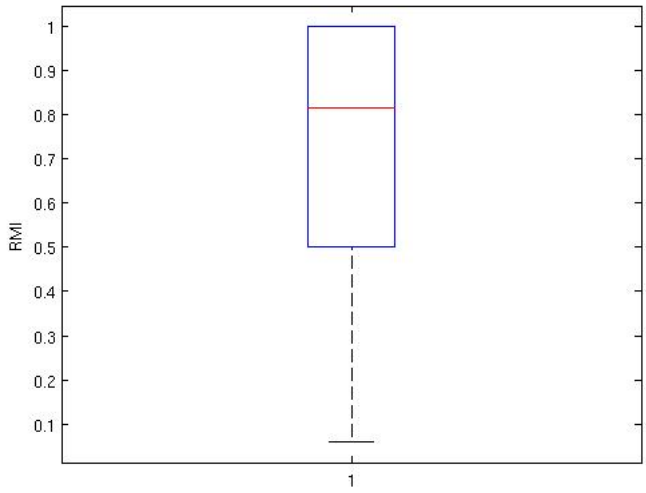

Figure 6.1: Box Plot for RMI in Carol 1.8.9.4

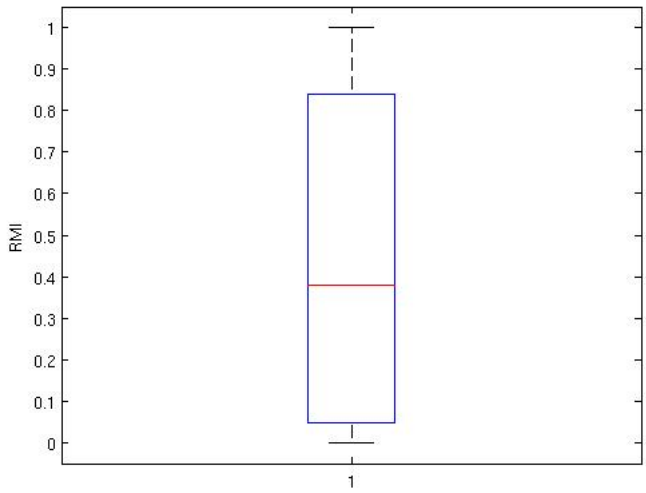

Figure 6.2: Box Plot for RMI in Tomcat 6.0.10

values of none of the packages (including the ones which underwent move method refactoring) standout when compared to the rest of the packages.

Based on the observations, a high value of RMI may not be a good candidate for bad smell that suggests move method refactoring. First of all, for move method refactorings within the same package and move method refactorings to a new package, RMI cannot be used as a bad smell metric. Secondly, our observations indicate that the packages which underwent refactorings did not have the maximum RMI value among all packages and there were many packages which had larger RMI value than the package in question. Even though the RMI of the packages were greater than average RMI, the difference was much less than standard deviation.

2. Of the 16 classes which underwent move method refactorings, 10 classes had a LCOM value of **0**. For the 6 classes which had a non-zero LCOM value, we compared the LCOM value with the average LCOM value computed across all classes in the version. The LCOM value of all the 6 classes was higher than average, however there were several classes in each of the versions which had much higher LCOM value than the 6 classes in question.

   Based on the observations, LCOM metric cannot be reliably used as an indicator of classes which require refactoring. Almost 62.5 % of the classes that underwent refactorings had an LCOM value of 0, this makes it almost impossible to consider LCOM as a bad smell for move method refactoring.

3. Of the 16 classes which underwent move method refactorings, 6 classes had CBO values less than average while 10 classes had CBO values greater than average. We also analyzed the boxplots of CBO values of all classes in **CAROL 1.8.9.4** and **TOMCAT 6.0.10**. The boxplots are illustrated in Figures 6.3 and 6.4 respectively. Analyzing the boxplot for **CAROL 1.8.9.4** shows that the CBO of the class which underwent refactoring is not an outlier. On the other hand, analysis of the boxplot for **TOMCAT 6.0.10** shows that the CBO of the class which underwent refactoring is indeed an outlier. However, there are a large number of classes in **TOMCAT 6.0.10** whose CBO values are outliers. Based on the observations, CBO of a class cannot be reliably used as a bad smell to identify the class as a candidate for move

method refactoring.

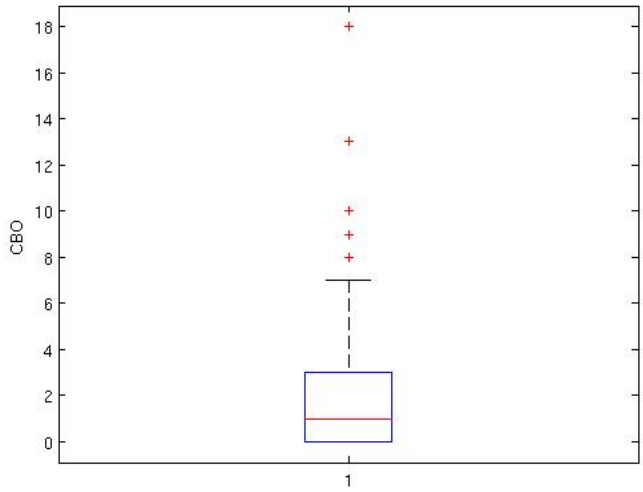

Figure 6.3: Box Plot for CBO in Carol 1.8.9.4

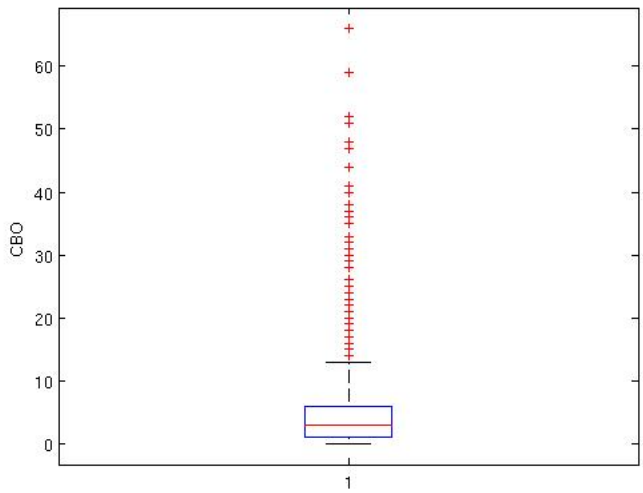

Figure 6.4: Box Plot for CBO in Tomcat 6.0.10

4. Of the 16 classes which underwent move method refactorings, 5 classes had

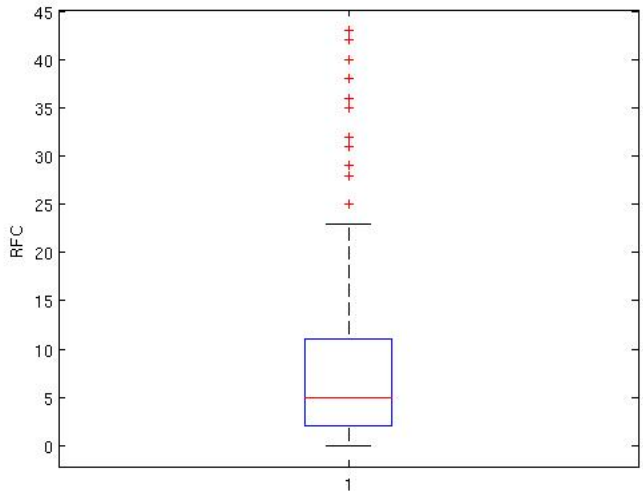

Figure 6.5: Box Plot for RFC in Carol 1.8.9.4

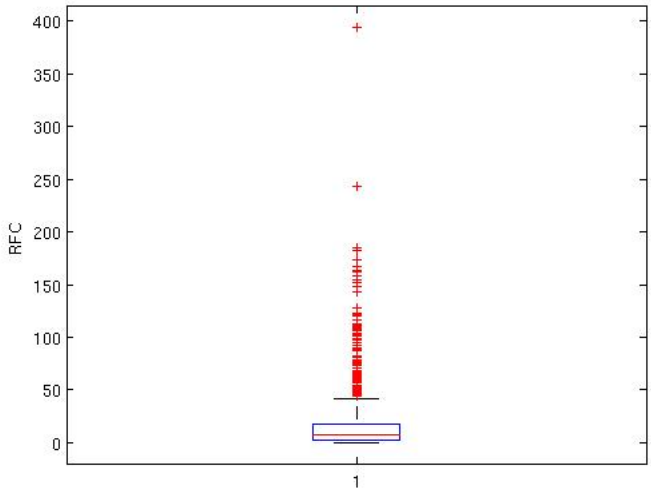

Figure 6.6: Box Plot for RFC in Tomcat 6.0.10

RFC values less than average while 11 classes had RFC values greater than average. We also analyzed the boxplots of RFC values of all classes in **CAROL 1.8.9.4** and **TOMCAT 6.0.10**. The boxplots are illustrated in Fig-

OSS Version	Observation
CAROL 1.2	The 2 classes which underwent refactorings showed an an increase in the Depth of Inheritance. The heuristics will not identify the refactoring
CAROL 1.5.2	The number of methods in both the classes which underwent refactoring showed an increase. Refactoring cannot be identified by the heuristic
CAROL 1.8.9.4	The depth of inheritance of the the class that underwent refactoring showed an increase. Refactoring cannot be identified by the heuristic
CAROL 2.0.8	The heuristics are able to identify the refactoring
CAROL 2.2.9	The heuristics are able to identify the refactoring
DNS 1.6.6	The old class does not exist anymore in the new version Refactoring cannot be identified by the heuristic
STRUTS 1.1	The heuristics are able to identify the refactoring in one of the classes. The second class showed an increase in the number of methods causing the heuristics to fail
STRUTS 1.2.7	The heuristics are able to identify the refactoring
TOMCAT 6.0.4	The number of methods in both classes that underwent refactoring did not change causing the heuristics to fail
TOMCAT 6.0.10	The depth of inheritance of the class that underwent refactoring increases causing the heuristics to fail

Table 6.6: Finding Moved Methods Via Change Metrics

ures 6.5 and 6.6 respectively. Interestingly, the classes which underwent refactoring in **CAROL 1.8.9.4** and **TOMCAT 6.0.10** both have RFC values that are outliers. However, it is still difficult to conclude anything definite about RFC value of a class being a bad smell.

6.4 Finding Refactorings via Change Metrics

In [DDN01], the authors define the following heuristic to identify a move method refactoring:

- The number of methods in a particular class should decrease and both the depth of inheritance and the number of children of the class should not change.

Version	RMI rate	LCOM rate	CBO rate	RFC rate
CAROL 1.2	1.998	0	1	0.5
CAROL 1.5.2	0.054	∞	0.846	0.967
CAROL 1.8.9.4	-0.043	0	0.8	0.057
CAROL 2.0.8	0.073	0	0.4	0.086
CAROL 2.2.9	2.102	0	0.230	0.878
DNSJAVA 1.6.6	-0.366	-0.403	-0.214	-0.164
STRUTS 1.1	0.318	0	0.292	0.696
STRUTS 1.2.7	2.36	-0.049	0.2	0.152
TOMCAT 6.0.4	0	0	0.647	-0.018
TOMCAT 6.0.10	0.010	-0.021	0.6	-0.329

Table 6.7: Move Method Refactoring Impact

We studied the change in various class-level metrics and applied the above heuristic for finding refactorings. The observations of our study are described in Table 6.6. The observations show that heuristics cannot identify refactorings on 10 out of 16 classes that underwent move method refactorings. The very low success rate (37.5 %) of change metric heuristics in [DDN01] indicate that it cannot be reliably used to find refactorings.

6.5 Refactoring Impact

In [HMKI08], the authors tried to estimate the impact of refactoring by quantifying the change in coupling between classes and cohesion of each class. To estimate the impact of move method refactoring, we studied the change rate of RMI, LCOM, CBO and RFC metrics. The results of our study is illustrated in Table 6.7. The change rate of a package level metric MET is calculated as follows:

Definition 6.1. Given packages A, B, C, D which underwent refactoring and packages A', B', C', D' into which methods were moved. Change rate in metric MET is defined as:

$$\text{MET change rate} = \frac{\left(\sum_{y \in A',B',C',D'} MET(y) - \sum_{x \in A,B,C,D} MET(x)\right)}{\left(\sum_{x \in A,B,C,D} MET(x)\right)}$$

The change rate of a class level metric MET is calculated as follows:

Definition 6.2. Given classes A, B, C, D which underwent refactoring and classes A', B', C', D' into which methods were moved. Change rate in metric MET is defined as:

$$\text{MET change rate} = \frac{\left(\sum_{y \in A', B', C', D'} MET(y) - \sum_{x \in A, B, C, D} MET(x) \right)}{\left(\sum_{x \in A, B, C, D} MET(x) \right)}$$

Studying the results from Table 6.7, it is clear that change rate in LCOM cannot be used to study the impact of move method refactoring since in most cases there is no change in LCOM. **CAROL 1.5.2** is an interesting test case where LCOM was initially zero and increases upon refactoring causing the change rate to become ∞.

On the other hand, the study shows that the move method refactoring does impact instability (RMI) of a package. However, in most cases (7 out of 10), the change rate of RMI is positive and in some cases quite high. Assuming that designers performed refactoring to improve overall code quality, it is surprising to see move method refactoring having a supposedly negative impact based on change rate in RMI. This suggests that we may not be able to estimate the impact of move method refactoring by studying the changes in metrics.

Of the 10 versions which underwent refactorings, only the refactoring applied on **DNSJAVA 1.6.6** shows a positive impact for both CBO. For the rest 9 testcases, CBO change rate is positive showing an increase in coupling after refactoring. **DNSJAVA 1.6.6** is an interesting testcase in that the **From** class gets deleted in the new version. With regard to the change rate in RFC, 3 of the move method refactorings show a positive impact while the rest 7 show an increase in RFC after refactoring. Of the 3, one of the of the versions that shows a positive impact is **DNSJAVA 1.6.6** which had the **From** class deleted.

Chapter 7

Pulled Up Method Refactoring

In this chapter, we will analyze the results obtained using the experimental setup described in Chapter 5 to study the relationship between metrics and pulled up method refactorings.

RefactoringCrawler was able to identify "PulledUp Method" refactorings on 7 versions out of 136 total versions analyzed. Table 7.1 provides details of the versions that had refactorings applied. The first column describes the version, the second column describes number of classes (NOC), the third column describes number of classes with pulledup method refactorings and the fourth and fifth column provides similar information for the number of packages (NOP).

This chapter is organized as follows. In Section 7.1, we will describe how we

OSS Version	**NOC**	**NOC with Pulled Up Methods**	**NOP**	**NOP with Pulled Up Methods**
CAROL 1.8.9.4	161	5	24	1
CAROL 1.8.9.5.4	163	5	24	1
DNSJAVA 1.5.1	103	2	4	1
STRUTS 1.1	332	1	41	1
STRUTS 1.2.4	340	2	40	1
STRUTS 1.2.7	345	5	40	1
TOMCAT 6.0.7	1163	1	100	1

Table 7.1: Pulled Up Method Refactoring

classified the different pull up method refactorings detected by RefactoringCrawler. We detail our observations on the experimental results and our inferences based on the observations in Sections 7.2, 7.3, and 7.4. Section 7.2 tries to see whether metrics can enable us identify good pulled up method refactoring opportunities. Section 7.3 analyzes whether change metrics can be reliably used to identify pulled up method refactorings. Section 7.4 answers the question on whether metrics can be used to evaluate the impact of pulled up method refactorings.

7.1 Pulled Up Method Classification

Classification	Description
PullUp 1	Existing superclass with multiple subclasses. Methods with identical results in the subclasses are pulled up into the superclass
PullUp 2	Multiple classes implementing the same interface. A new superclass is created to implement the interface and the existing classes are made subclasses of the new superclass. Methods with identical results in subclasses are then pulled up into the superclass
PullUp 3	Existing superclass with multiple subclasses such that there are two subclasses that have methods with identical results. A new subclass is created and the two nearly identical subclasses are made subclasses of the newly created subclass. Methods with identical results are then pulled up into the newly created subclass
PullUp 4	Existing superclass with multiple subclasses such that the superclass and the subclass are in different packages. Methods are pulledup from a particular subclass to the superclass
PullUp 5	Two classes implementing the same interface. One class is made the subclass of another and methods pulled up from the subclass to the superclass

Table 7.2: Pulled Up Method Classification

An interesting observation from the experimental data is the existence of refactorings where methods from a single class have been pulled up. This is in contrast with the typical pull up method refactoring description where methods with identical results on multiple subclasses are pulled up into the superclass. This

prompted us to study the nature of pulledup method refactorings identified by RefactoringCrawler. The pulledup method refactorings identified by RefactoringCrawler can be classified into 5 different categories as enumerated in Table 7.2. The following observations can be made from the 5 different "Pulled Up Method" categories listed in Table 7.2.

- Creation of the superclass and pulling up of methods from subclasses to the superclass is listed as a single atomic operation in Pulled up methods **PullUp 2** and **PullUp 3**. This is partly due to the fact that our analysis is at the granularity of tagged releases. However, for **CAROL 1.8.9.5.4** which had a **PullUp 2** refactoring applied on it, the tagged releases were consecutive SVN versions. Consecutive SVN revisions is the lowest level of granularity achievable when studying software evolution on an existing open source code.

- **PullUp 4** maybe better classified as a move method since its aim is not to remove similar methods from subclasses, but to pull up a method to improve coupling and cohesion. It is also the only pulled up method category that involves methods moving from one package to another. Typically the pulled up methods remain in the same package.

7.2 Bad Smells to Identify Refactoring Opportunities

Duplicated Code [Fow00] is bad smell that indicates a pull up method refactoring opportunity. In this case, identical or very similar code exists in two sibling subclasses. Duplicated code removal reduces the size of code thereby making it easier to understand, and hence, easier to modify. It also reduces the possibility of bugs being introduced due to inconsistent modifications in the duplicated code fragments. Except for **PullUp 4**, rest of the "PullUp Method" refactoring categories enumerated in Table 7.2 exhibit the *duplicated code* bad smell. However, none of the existing analytical or descriptive object-oriented metrics that we described in Chapter 4 can be reliably used to identify code clones in the design.

A large Depth of Inheritance (DIT) metric value for a class is typically considered a bad smell since it implies increased difficulty in testing and decreased comprehensibility for the particular class. Pulled Up method refactoring is the suggested refactoring to overcome this bad smell and improve the code. Of the 21 classes which underwent pulled up method refactoring, 15 classes had a DIT value of **1**, the rest had a DIT value of **2**. This implies that DIT value could not have been used as a reliable metric to identify classes which needs pulled up method refactoring.

7.3 Finding Refactorings via Change Metrics

Demeyer et al. [DDN01] proposed several different heuristics to detect refactorings by applying lightweight object-oriented metrics to successive versions of a software system. The heuristics to identify refactorings relied on three main class level metrics; **(1)** Depth of Inheritance (DIT), **(2)** Number of Methods (NOM), and **(3)** Number of Children (NSC). Of the 6 different heuristics proposed in [DDN01], the following 3 heuristics were found applicable in our analysis.

- *Split into Superclass*: This heuristic searches for refactorings that optimize the class hierarchy by splitting functionality from a class into a newly created superclass. Essentially, the heuristic looks for the creation of a superclass, together with a number of pulled up methods.

- *Merge with Subclass*: This heuristic – like the one above – searches for refactorings that optimize the class hierarchy. The heuristic essentially looks for the removal of a subclass, together with a number of pulled up methods from the subclass to the superclass. The dual of this heuristic is called *Split into Subclass*, where a subclass is created and methods are pushed down to the subclass from the superclass

- *Move to Superclass*: This is yet another heuristic that is applicable in our analysis to detect pulled up refactorings using change metrics. The heuristic looks for DIT and NSC to remain the same along with a reduction in NOM.

OSS Version	Observation
CAROL 1.8.9.4	All 5 classes showed a decrease in Number of Methods (NOM) and an increase in the Depth of Inheritance (DIT). According to heuristics defined in [DDN01] to find refactorings, this would correspond to *Split into SuperClass* where methods are moved from subclasses to the newly created Superclass. The applied refactoring confirms the hypothesis.
CAROL 1.8.9.5.4	All 5 classes showed a decrease in Number of Methods and and an increase in the Depth of Inheritance. The change in metrics would suggest a *Split into SuperClass* and the applied refactoring confirms the hypothesis.
DNSJAVA 1.5.4	Methods were pulled up from two classes and the classes show a decrease in NOM and increase in DIT. *Split into SuperClass* is confirmed.
STRUTS 1.1	A single method was pulled up from a sub class into a superclass. The class shows a decrease in number of methods and an increase in the number of children (NSC). DIT remains the same. Change in metrics would suggest a *Split into SubClass*, however the refactoring applied is essentially a *Move to SuperClass*
STRUTS 1.2.4	Two methods with identical results from two subclasses were pulled into superclass. NOM decreases for both classes. For one of the classes DIT and NSC remains the same while for the other there is an increase in NSC. Change in metrics would suggest a *Move to SuperClass* for the first class and a *Split into SubClass* for the second one. Heuristics correctly identify the first refactoring and incorrectly classifies the second
STRUTS 1.2.7	All 5 classes which underwent pull-up refactoring had an increase in NOM as well as an increase in DIT. Change in metrics would not infer any refactorings where a *Split into SuperClass* would have been the correct inference
TOMCAT 6.0.7	There were two identical subclasses and one subclass was made the subclass of another. The class from which methods were pulled up had a reduction in NOM and increase in DIT. Change metrics would infer a *Split into SuperClass* which is confirmed.

Table 7.3: Finding Pulled Up Methods Via Change Metrics

OSS Version	Old Sum (From)	Old Sum (To)	New Sum (From)	New Sum (To)
CAROL 1.8.9.4	0	-	0	0
CAROL 1.8.9.5.4	3.64	-	0.25	0.83
DNSJAVA 1.5.1	1.15	-	0	1.38
STRUTS 1.1	0	0.93	0	0.93
STRUTS 1.2.4	0.68	0.8	0.67	0.82
STRUTS 1.2.7	3.85	-	4.19	0.58
TOMCAT 6.0.7	0.95	0.94	0	0.95

Table 7.4: PulledUp Method Refactoring – LCOM Data

We studied the change in various class-level metrics and applied the heuristics for finding refactorings described in the work by Demeyer et al. [DDN01]. The observations of our study are described in Table 7.3.

Based on the observations detailed in Table 7.3, out of 21 classes which underwent pull up method refactoring, heuristics for identifying refactorings based on change metrics [DDN01] can correctly identify the proper refactorings for 13 classes. In **STRUTS 1.2.7**, heuristics cannot identify the *Split into Superclass* refactoring performed on the all the 5 classes that underwent refactoring (false negative). In **STRUTS 1.1** and **STRUTS 1.2.4**, heuristics incorrectly classify *Move to Superclass* refactorings as *Split into Subclass* refactorings.

7.4 Refactoring Impact

Estimating the impact of pulled up method refactoring using metrics has not been studied before. In [HMKI08], the authors allude to the fact that the impact of pulled up method refactorings could possibly be studied using metrics, but the case study only refers to move method refactoring. Pulled up method refactoring mainly involves moving of methods within the same package and hence does not have any impact on instability of a package. Thus studying change rate of package instability (RMI) metric will not be of any help in determining the impact of refactoring. Since pulled up method refactoring involves finding code clones and getting rid of them, we decided to study the change rate in weighted methods per class (WMC)

OSS Version	Old Sum (From)	Old Sum (To)	New Sum (From)	New Sum (To)
CAROL 1.8.9.4	282	-	66	68
CAROL 1.8.9.5.4	136	-	104	14
DNSJAVA 1.5.1	54	-	12	52
STRUTS 1.1	11	84	6	57
STRUTS 1.2.4	31	15	19	20
STRUTS 1.2.7	136	-	221	12
TOMCAT 6.0.7	65	123	0	97

Table 7.5: PulledUp Method Refactoring – WMC Data

OSS Version	Old Sum (From)	Old Sum (To)	New Sum (From)	New Sum (To)
CAROL 1.8.9.4	24	-	23	5
CAROL 1.8.9.5.4	28	-	33	2
DNSJAVA 1.5.1	24	-	16	14
STRUTS 1.1	6	3	5	3
STRUTS 1.2.4	8	3	8	3
STRUTS 1.2.7	4	-	13	0
TOMCAT 6.0.7	13	16	1	36

Table 7.6: PulledUp Method Refactoring – CBO Data

in addition to the studying the change rate in lack of cohesion of methods (LCOM), coupling between objects (CBO), and response for a class (RFC) metrics as defined in Section 6.5. The change rate in WMC is calculated as follows:

Definition 7.1. Given classes A, B, C, D which underwent refactoring and classes A', B', C', D' into which methods were pulled. Change rate in WMC is defined as:

$$\text{WMC change rate} = \frac{\left(\sum_{y \in A',B',C',D'} WMC(y) - \sum_{x \in A,B,C,D} WMC(x)\right)}{\left(\sum_{x \in A,B,C,D} WMC(x)\right)}$$

To study the change rate of each of the four metrics, we calculated the sum of each of these metrics over the classes which underwent refactoring and classes into which methods were pulled into. The data for each of the four metrics is presented in Tables 7.4, 7.5, 7.6, and 7.7. In each of the four tables, the first column

OSS Version	Old Sum (From)	Old Sum (To)	New Sum (From)	New Sum (To)
CAROL 1.8.9.4	183	-	42	41
CAROL 1.8.9.5.4	73	-	60	11
DNSJAVA 1.5.1	65	-	10	41
STRUTS 1.1	14	49	11	49
STRUTS 1.2.4	33	16	42	20
STRUTS 1.2.7	74	-	99	8
TOMCAT 6.0.7	50	62	0	46

Table 7.7: PulledUp Method Refactoring – RFC Data

Version	LCOM rate	WMC rate	CBO rate	RFC rate
CAROL 1.8.9.4	0	-0.524	0.167	-0.546
CAROL 1.8.9.5.4	0.208	-0.182	0.25	-0.027
DNSJAVA 1.5.1	0.2	0.185	0.25	-0.215
STRUTS 1.1	0	-0.021	-0.111	-0.47
STRUTS 1.2.4	0.013	-0.188	0	0.292
STRUTS 1.2.7	0.841	0.713	2.25	0.445
TOMCAT 6.0.7	-0.497	-0.484	0.275	-0.589

Table 7.8: Pulled Up Method Refactoring Impact

illustrates the open source project version which underwent refactoring. We refer to the version in which refactoring was detected to be the old version and the version after refactoring was applied as the new version. The second column depicts the sum of the particular metric over all classes that underwent refactoring (referred to as **From** in the Table) in the old version. The third column column depicts the sum of the particular metric over all classes into which methods were pulled into (referred to as **To** in the Table) in the old version. The fourth and fifth column depicts the sum of the particular metric in the new version over **From** and **To** classes respectively. In the Table, "-" refers to the fact that there was no data. In particular, out of the 7 pulled up method refactorings detected, in 4 of the refactorings the superclass into which methods were pulled into did not exist in the old version.

The results of our study is illustrated in Table 7.8. The results show that the

change rate in LCOM maybe better suited to study the impact of pulled up method refactoring as opposed to move method refactoring. There are only two versions where there is no change in LCOM. However, except for **TOMCAT 6.0.7**, the other versions show an increase in LCOM suggesting negative impact for refactoring.

Given that pulled up method refactoring should get rid of code clones, one would guess that WMC change rate would always be positive indicating the positive impact of pulled up method refactoring in reducing code complexity. However, there are two versions (**DNSJAVA 1.5.1** and **STRUTS 1.2.7**) which show an increase in the change rate of WMC.

The change rate of CBO shows an interesting trend of almost always having a negative impact, i.e., coupling between classes increase after pulled up method refactoring. This shows that although pulled up method refactoring is able to remove duplicated code, by moving functionality from subclasses to superclasses, it could increase the coupling of the superclasses while making little change to the coupling of the subclasses.

On the other hand, change rate of RFC shows a positive impact on 5 out of 7 versions. This is because by pulling up methods from multiple subclasses, we are enabling a big reduction in the cumulative RFC of the set of the classes which underwent refactoring.

7.4.1 Summary

Given that there are very few refactorings identified (7 out of 136 versions analyzed) and assuming that the ultimate aim of designers was to improve code quality through refactorings, it is surprising that a large percentage of refactorings appear to have a negative impact. One of the main aims of refactoring impact estimation is to identify the refactorings performed in older versions that had the highest impact and use that information when making a decision to apply refactoring on a later version. The study shows that either analyzing the change rate of metrics may not be well suited to estimate refactoring impact or the applied refactorings resulted in a negative impact even though the designers may have applied them to improve software quality.

Chapter 8

Pushed Down Method Refactoring

In this chapter, we aimed to study the relationship between metrics and pushed down method refactorings based on the experimental results obtained using the analysis described in Chapter 5. Pushed down method refactoring essentially moves a method from a superclass to one of the subclasses [Fow00]. Subclasses inherit methods and features from their parent class. Thus, pushed down method refactoring is applicable when methods in the superclass are only applicable to some of its subclasses. Pushed down refactoring enables the superclass to only hold what is common among all the subclasses. *Refused Bequest* [Fow00] is bad smell that indicates a push down method refactoring opportunity. A base class inherits methods that are not used at all. It indicates poor inheritance design. A possible scenario described by Fowler, is when the subclass is reusing base class behavior but does not want to support the interface of the superclass.

8.1 Experimental Results and Conclusion

RefactoringCrawler [DCMJ06] was not able to identify pushed down method refactorings in any of the 136 versions of the open source projects analyzed. This could imply one of three possibilities:

1. Pushed down method refactoring is not a commonly applied refactoring.
2. The 4 different open source projects and their respective versions we analyzed are not good representatives of all refactorings we attempted to study.
3. RefactoringCrawler is not adept at detecting push down method refactorings.

Chapter 9

Refactoring Impact on Software Quality

Software undergoes frequent modifications, improvements and enhancements to cope with the changes in business requirements [KL03]. These changes applied over time can have an adverse impact on the quality of software. The design could become complex, difficult to maintain and test, and easily susceptible to bugs. Refactoring [Fow00] is one of the most important and commonly used techniques to improve the quality of the code by eliminating redundancy and reducing complexity; frequently refactored code is believed to be easier to understand, maintain and test.

Software metrics provides an easy means to extract useful and measurable information about the structure of a software system. Metrics have also proved to reflect software quality and has been widely used in evaluating software quality [BJM76]. Several researchers have also successfully correlated metrics with quality [SK03, SAOB02].

Several tools are available for collecting software metrics as well as for performing automated refactoring on the source code. Given the abundance of such tools, one would expect designers to aggressively use refactoring to improve software quality. Several researchers have attempted to study the impact of refactoring on software quality as measured in terms of metrics [SS07, HMKI08]. In [SS07],

the authors extract version control system logs of popular open source software systems to detect changes marked as refactorings and examine how software metrics are affected by this process. However, the authors only analyze metrics of the methods/classes/packages affected by refactoring. Again, in [HMKI08], the authors only analyze the change rate of metrics of the classes affected by refactoring. We also performed a similar analysis which is described in detail in Chapters 5, 6, and 7.

In this chapter, instead of just studying the change in metrics of the classes affected by refactoring, we will study the temporal evolution of various software metrics. There are several reasons for the nature of our study. First of all, we are only analyzing open source projects at the granularity of releases. When we detect a refactoring in a particular class/package, that may not be the only change occurring in the class/package. This implies that refactoring may not be the only cause for the changes in metrics and results obtained from analyzing the changes in metrics may not give us the complete picture on the impact of refactoring on software quality.

Assuming that the detected refactoring is only one among many changes applied by the designers to improve quality of the design, we make the following research hypotheses. Given two releases of the open source project: old and new. Refactoring was detected between the old and new release with the new release consisting of refactored code.

- **Hypothesis 1:** The quality of the system as measured using software metrics was very poor in the old release.
- **Hypothesis 2:** The quality of the system as measured using software metrics should improve in the new release.

Of the 4 open source projects we had analyzed, **CAROL** had the largest number of versions as well as the largest number of refactorings detected. In this chapter, we will restrict our analysis on **CAROL**. We use the metrics calculated by the Eclipse Metrics plug-in 1.3.6 [Sua] for our software quality analysis. For detailed results from the other 3 open source projects, please refer to the Appendix.

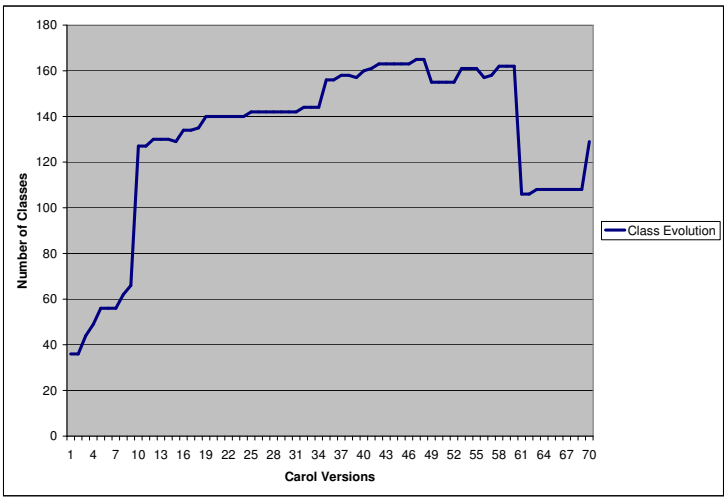

Figure 9.1: Evolution of Number of Classes in Carol

9.1 Refactoring Impact

According to the laws of software evolution [LPR+97], modern software systems exhibit the following characteristics.

- Software systems must continuously undergo changes or else they will become progressively unsatisfactory (law 1)
- As a software system evolves, its complexity increases unless work is done to maintain it or reduce it (law 2)
- Quality of the software system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes (law 7)

To study the evolution of open source projects that we analyzed, we plotted the evolution of total number of classes in the project. The evolution of the total number of classes in **CAROL** is depicted in Figure 9.1. We can make the following observations by studying the evolution of total number of classes. First of all, there is a

continuous change in the total number of classes (except for a few intermediate versions). After the initial couple of versions, there is a big increase in the number of classes (most probably corresponds to addition of new functionality to the software system), followed by changes at regular intervals. Around 3/4th of the lifetime, there is a huge drop in the number of classes followed by the number of classes remaining steady for a while and finally yet another increase in the number of classes at the end. Even though the changes in total number of classes in the project is a very crude approximation of continuous changes in a software system, the changes in the number of classes with each version can be considered a confirmation of the fact that **CAROL** does obey **law 1** described above.

To study how complexity and quality of the software system evolves over time, we study the following three metrics, **(1)** Instability (RMI), **(2)** Weighted Methods per Class (WMC), and **(3)** Lack of Cohesion of Methods (LCOM). To better illustrate the impact of refactoring, in the figures that are used to depict the evolution of each metric, we identify the versions at which we detected refactoring through straight lines that intersect the graph with the X-axis. After the removal of false positives, we detected refactorings in 6 versions of **CAROL**. Of the 6 versions, the first 2 versions only underwent move method refactorings, the third version underwent move method refactoring and pulled up method refactoring, the fourth version only underwent pulled up method refactoring, and the fifth and sixth versions only underwent move method refactorings.

1. Instability (RMI): RMI is a package level metric. It is an indication of a package's resilience to change. It has a range [0,1] where, 0 indicates a maximally stable package and 1 indicates a maximally unstable package. For the purpose of our study, we calculated the average RMI of the whole system (average across all packages in the system). The evolution of average RMI across all packages in **CAROL** is illustrated in Figure 9.1. The average RMI for **CAROL** starts relatively high, increases steadily and reaches a peak. The peak is followed by a sharp decline, followed by a further rise and then RMI decreases steadily for several versions and shows another sharp increase at the end.

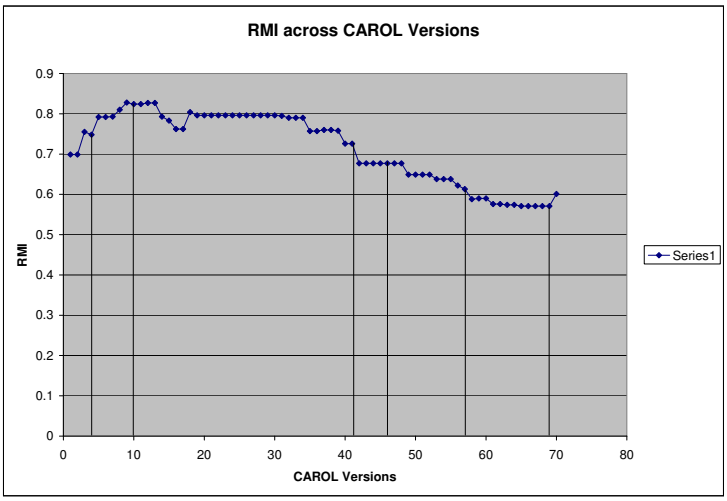

Figure 9.2: Temporal Evolution of RMI for Carol

RMI is more relevant to move method refactoring as opposed to pulled up method refactoring, hence we only need to analyze it for refactorings 1 to 3 and 5 to 6. With regard to our two hypotheses described earlier in the chapter, it is not the case that the version at which refactoring has been applied is a local maxima; i.e., the version at which refactoring happened is not really the one with poorest quality among all neighboring versions. For none of the move method refactorings, we detected the condition that the old release had a relatively high average RMI. With regard to the second hypothesis, only the third move method refactoring (**CAROL 1.8.9.4** shows a decrease in RMI after refactoring. Coincidentally, this is also the only move method refactoring that involved moving methods from one package to a different existing package. Since other move method refactorings involved moving classes within the same package or from one package to a newly created package, it seems plausible that the average RMI may not show a big decline after move method refactoring.

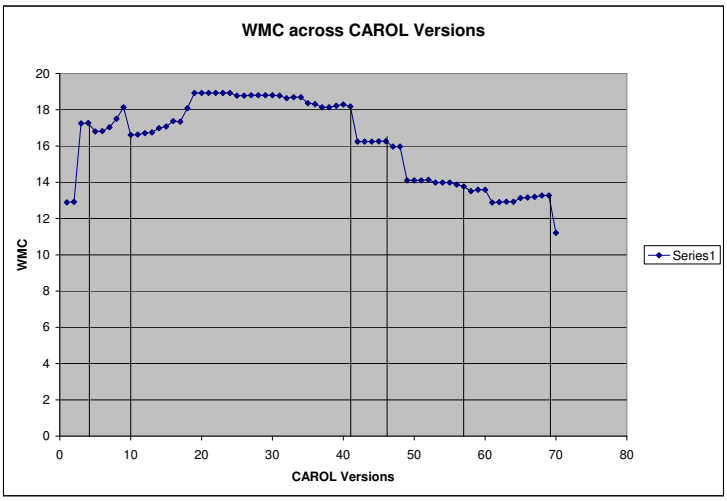

Figure 9.3: Temporal Evolution of WMC for Carol

2. Weighted Methods per Class (WMC): WMC is a class level metric. It is obtained by computing the sum of the McCabe Cyclomatic Complexity for all methods in a class. Large values of WMC indicates complex classes. For our study, we calculate the average WMC of the whole system (average across all classes in the system). The evolution of average WMC across all classes in **CAROL** is illustrated in Figure 9.1. WMC shows a similar evolution pattern as RMI. It starts quite low and shows a big increase in the early stages of the software design cycle. WMC stays steady for a while and then shows a consistent decline with each advancing version.

WMC is a better indicator of impact of pulled up method refactoring as opposed to move method refactoring. Thus we need to only analyze the 3rd and 4th refactorings depicted in the Figure. The values of WMC at the versions at which refactorings was applied do not confirm hypothesis 1. However, there is indeed a decrease in WMC after refactoring which supports hypothesis 2.

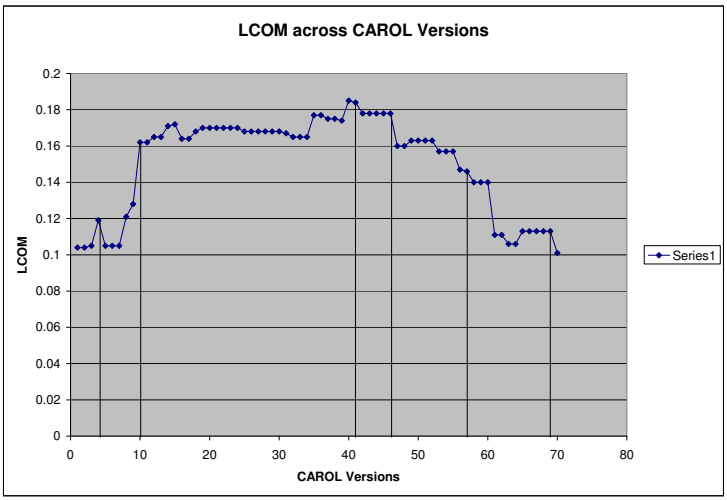

Figure 9.4: Temporal Evolution of LCOM for Carol

3. Lack of Cohesion of Methods (LCOM-HS): LCOM is a class level metric that computes cohesiveness of a class [HS96]. A high LCOM value indicates decreased encapsulation and increased complexity while a low value implies high cohesion and good design. For our study, we calculate the average LCOM of the whole system (average across all classes in the system). The evolution of average LCOM across all classes in **CAROL** is illustrated in Figure 9.1. LCOM shows a very interesting evolution pattern. It starts quite low and is followed a big increase. LCOM remains steady for quite a while after the big increase, increases again, followed by a steady decline till the end.

LCOM is a good indicator of the impact of both move method and pulled up method refactorings. The values of WMC at the versions at which refactorings was applied do not confirm hypothesis 1. However, in all the cases, there is indeed a decrease in average LCOM after refactoring which supports hypothesis 2. The biggest decrease in LCOM is after the second pulled up method refactoring (**CAROL 1.8.9.5.4**).

9.2 Conclusion

In conclusion, the study of evolution of software using various software metrics confirms the laws of software evolution [LPR+97] described in Section 9.1. We studied the evolution of 3 different software metrics; Instability (RMI), Weighted Methods per Class (WMC) and Lack of Cohesion of Methods (LCOM). The study illustrated how complexity increases and the quality decreases as the software evolves through different versions. The study also illustrated the impact of refactoring on complexity and quality of software. Overall refactoring appeared to have a positive impact by reducing complexity and increasing the quality of software.

Chapter 10

Study Limitations

In this chapter, we will describe the limitations to our study.

Subject Programs and Versions: We studied 136 program versions covering 4 open-source projects. Although programs under consideration are all very different, we cannot possibly claim that their version histories are representative of all kinds of software projects.

Our study is restricted to analyzing released versions of these open source projects. Essentially, we are analyzing a cumulative set of refactorings, bug fixes, new features and quality, performance improvements. It is possible that we could miss refactorings as they could be masked by other changes.

Our study examined how the metrics of the open source projects under consideration were affected when certain refactorings were applied regardless of the reasons that led to the refactoring decision. It is very well possible that a refactoring decision was motivated by other considerations, performance for example, which might not necessarily improve the quality of the software system as measured by the metrics we included in our study.

External Tools: For our study, we have relied on the RefactoringCrawler tool to automatically detect refactorings in the program versions under consideration. There is a possibility that RefactoringCrawler could falsely detect refactoring in

versions where there were none. We analyzed every refactoring detected by RefactoringCrawler to weed out the false positives. It is also possible that RefactoringCrawler might have missed some refactorings. This could possibly explain why we detected very few refactorings even after analyzing 136 versions of open source projects.

Chapter 11

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss future research directions. The relationship between metrics and refactorings has been widely researched. However, there has not been a detailed simultaneous study of the various relationships on existing open source projects. Moreover, our research is qualitatively different from previous research in several different ways. Our analysis using automatic refactoring detection techniques is much more comprehensive than previous research which relied on refactorings detected using change logs. We are the first to compare automatic refactoring detection techniques that do not use metrics and refactoring detection techniques that use metrics. We are also the first to use an automatic refactoring detection tool to mine for refactorings in existing open source projects to analytically study the impact of refactoring on software quality as measured by metrics.

Metrics as hints for identifying refactoring opportunities:

We conclude from our study that there does not exist any particular set of metrics that can be reliably used to identify bad smells for pulled up method refactorings. For move method refactorings, even though several bad smells based on high values of certain metrics have been proposed, our study indicates that the metrics associated with particular bad smells cannot reliably identify source code whose complexity can be reduced through move method refactoring.

Finding Refactorings via Change Metrics:

We can conclude from our study that heuristics based on change metrics do a very poor job in identifying refactorings. The fact that refactoring is not the only change applied by the designers causes the heuristics to produce wrong results. Often they do not identify any refactorings or incorrectly classify the refactoring identified.

Metrics to study refactoring impact:

Our study shows that refactoring negatively impacts the software quality as measured by metrics. This could mean that either refactorings did not achieve what they were intended to achieve as indicated by the metrics or that the analysis of change rate of metrics gives a very bad estimate of the impact of refactoring in software quality.

Our analysis is further complicated by the fact that we are analyzing at the granularity of releases and there could a multitude of other changes interspersed with refactoring. We also analyzed the open source projects at the system level, by calculating an average of the metric values over all packages and classes. Our study of evolution of class and package level metrics confirmed the laws of software evolution [LPR+97]. Also, overall refactoring appeared to have a positive impact by reducing complexity and increasing the quality of software.

11.0.1 Future Work

There are numerous future work directions to enhance the results reported herein. First of all, in our work, we have only studied move method and pulled up method refactoring. It would be very interesting to study other refactoring techniques. Secondly, we would like to use a automatic refactoring detection technique other than RefactoringCrawler to detect refactorings. Thirdly, we would like to study more open source projects and if possible study them at the granularity of each SVN/CVS revision to detect refactorings.

Appendix A

Struts Quality Analysis

In this appendix, we will illustrate the evolution of RMI, LCOM and WMC metrics in Struts.

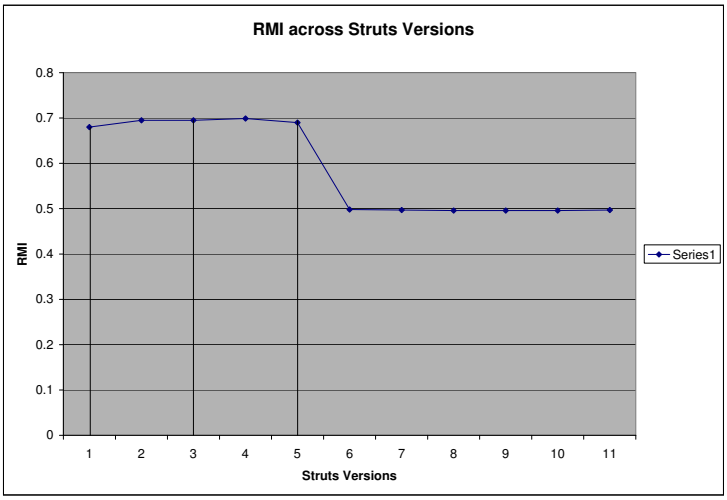

Figure A.1: Temporal Evolution of RMI for Struts

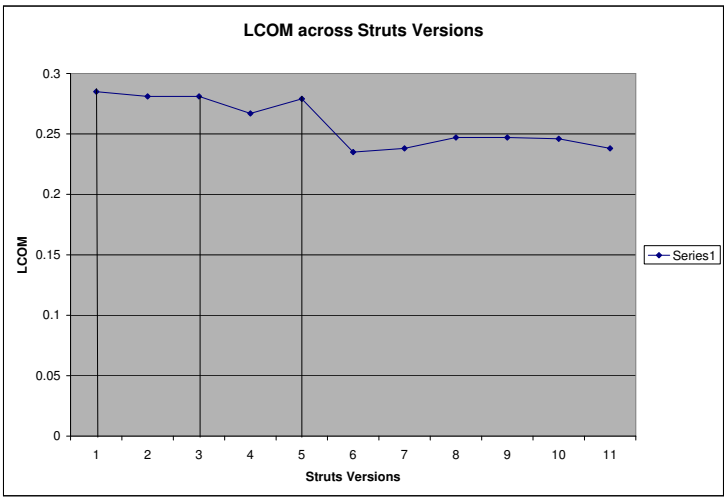

Figure A.2: Temporal Evolution of LCOM for Struts

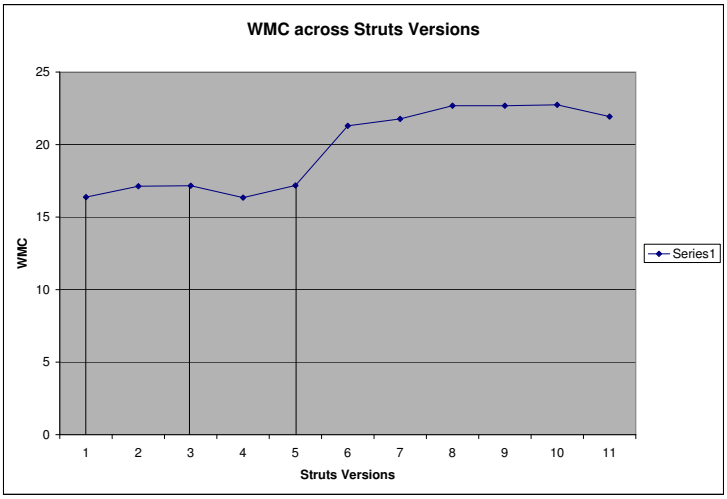

Figure A.3: Temporal Evolution of WMC for Struts

Appendix B

Dnsjava Quality Analysis

In this appendix, we will illustrate the evolution of RMI, LCOM and WMC metrics in Dnsjava.

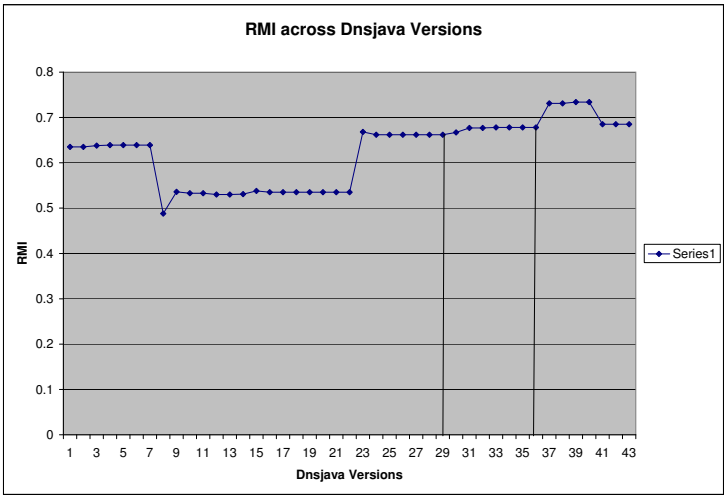

Figure B.1: Temporal Evolution of RMI for Dnsjava

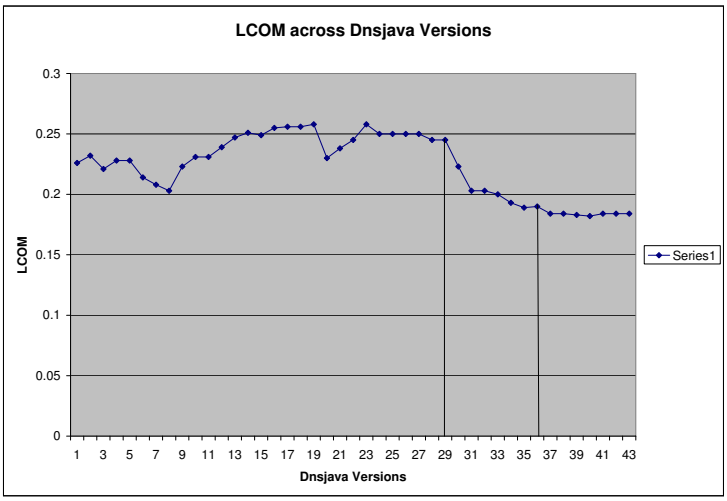

Figure B.2: Temporal Evolution of LCOM for Dnsjava

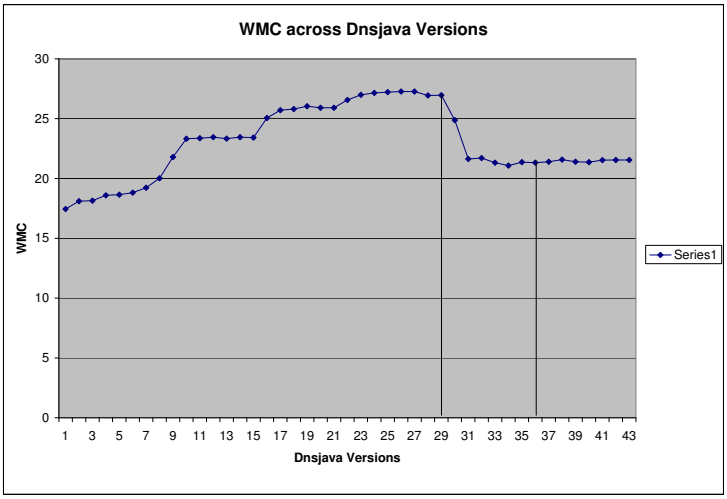

Figure B.3: Temporal Evolution of WMC for Dnsjava

Appendix C

Tomcat Quality Analysis

In this appendix, we will illustrate the evolution of RMI, LCOM and WMC metrics in Tomcat.

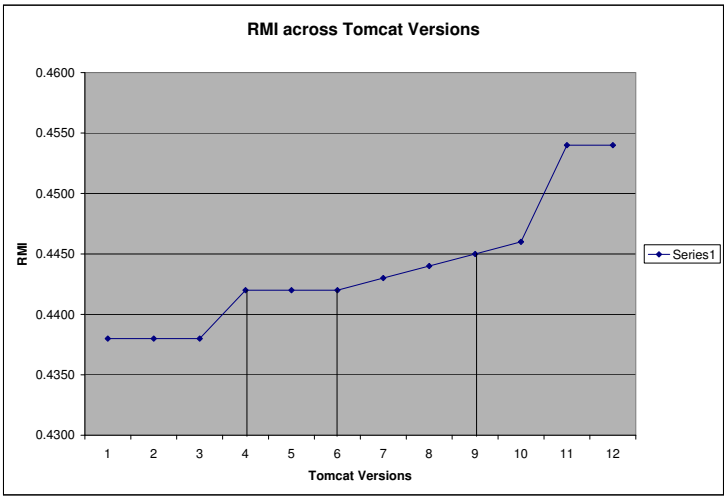

Figure C.1: Temporal Evolution of RMI for Tomcat

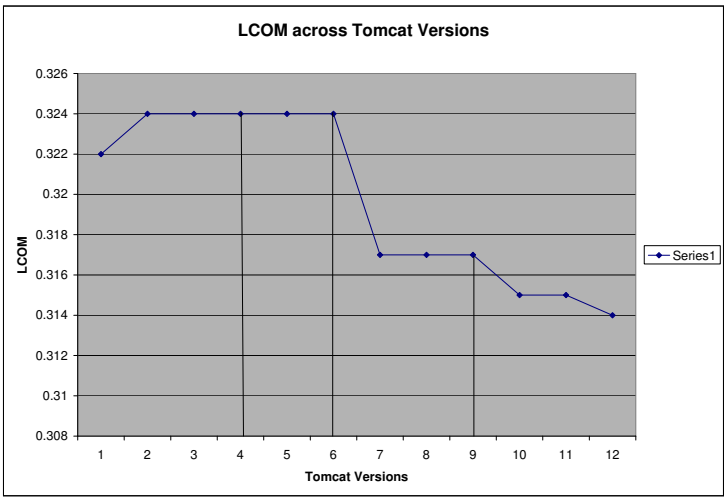

Figure C.2: Temporal Evolution of LCOM for Tomcat

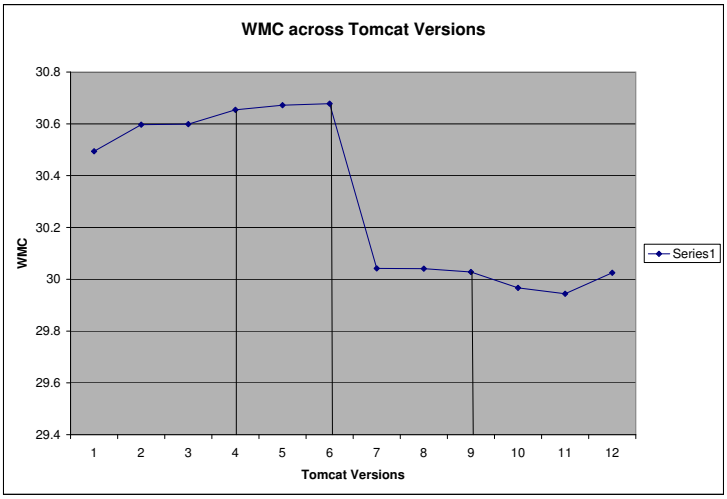

Figure C.3: Temporal Evolution of WMC for Tomcat

Bibliography

[BJM76] B.W.Boehm, J.R.Brown, and M.Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[BR89] Ted J. Biggersta and Charles Richter. Reusability framework, assessment, and directions. In *In Software Reusability - Volume I: Concepts and Models*, 1989.

[Bro95] Fred Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[CLMM05] Y. Crespo, C. Lpez, E. Manso, and Ral Marticorena. Language independent metric support towards refactoring inference. In *9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2005.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object Oriented Programming*, 2006.

[DDN01] Serge Demeyer, Stphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2001.

[EJ90] E.J.Chikofsky and J.H.Cross. Reverse engineering and design recovery: A taxonomy. In *IEEE Software*, 1990.

[FG06] B. Fluri and H. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[HD05] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of 27th International Conference on Software Engineering*, 2005.

[HGH08] A. Hindle, D.M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the 2008 International Workshop on Mining Software Repositories*, 2008.

[HMKI08] Yoshiki Higo, Yoshihiro Matsumoto, Shinji Kusumoto, and Katsuro Inoue. Refactoring effect estimation based on complexity metrics. In *Proceedings of the 19th Australian Software Engineering Conference*, 2008.

[HS96] Brain Henderson-Sellers. *Object Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[KL03] W. Kadir and P. Loucopoulos. Relating evolving business rules in software design. In *International Conference on Software Engineering and Practice (SERP)*, 2003.

[KNG07] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program veriosns. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[LPR+97] M. M. Lehman, D. E. Perry, J. F. Ramil, W. M. Turksi, and P. Wernick. Metrics and laws of software evolution - the nineties view. In *Fourth International Symposium on Software Metrics*, 1997.

[LPW97] L.Briand, P.Devanbu, and W.Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th International Conference in Software Engineering*, 1997.

[LSV99] L.Briand, S.Morasca, and V.R.Basili. Defining and validating measures for object-based high-level design. In *IEEE Transactions in Software Engineering*, 1999.

[Mar03] Robert C. Martin. *Agile Software Devlopment: Principles, Patterns and Practices*. Prentice Hall, 2003.

[M.H77] M.H.Halstead. Elements of software science. In *Operating and Programming Systems Series*, 1977.

[MJ94] M.Lorenz and J.Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[MT04] Tom Mens and Tom Tourw. A survey of software refactoring. In *IEEE Transactions on Software Engineering*, 2004.

[PD06] P.Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[RD03] F.V. Rysselberghe and S. Demeyer. Reconstruction of succesful software evolution using clone detection. In *Proceedings of 6th International Workshop on Software Principles of Software Evolution*, 2003.

[RSG08] J. Ratzinger, T. Sigmund, and H.C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Workshop on Mining Software Repositories*, 2008.

[SAOB02] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. In *Information Systems Journal*, 2002.

[SC91] S.R.Chidamber and C.F.Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the 6th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1991.

[SC94] S.R.Chidamber and C.F.Kemerer. A metrics suite for object oriented design. In *IEEE Transactions in Software Engineering*, 1994.

[SD81] S.M.Henry and D.G.Kafura. Software structure metrics based on information flow. In *IEEE Transactions in Software Engineering*, 1981.

[SD84] S.M.Henry and D.G.Kafura. The evaluation of software systems' structure using qualitative software metrics. In *Software- Practice and Experience*, 1984.

[SK03] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. In *IEEE Transactions on Software Engineering*, 2003.

[SS07] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *International Workshop on Software Quality*, 2007.

[SSL01] Frank Simon, Frank Steinbrckner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[Sua] Frank Suaer. Metrics 1.3.6 - getting started. `http://metrics.sourceforge.net/`.

[TA94] T.J.McCabe and A.H.Watson. Software complexity. In *Crosstalk, Journal of Defense Software Engineering*, 1994.

[TC89] T.J.McCabe and C.W.Butler. Design complexity measurment and testing. In *Communications of ACM*, 1989.

[T.J76] T.J.McCabe. A complexity measure. In *IEEE Transactions in Software Engineering*, 1976.

[VIZ] Vizzanalyzer, arisa, controlling software. `http://www.arisa.se/vizz_analyzer.php`.

[VM96] V.R.Basili and W.L. Melo. A validation of object-oriented design metrics as quality indicators. In *IEEE Transactions in Software Engineering*, 1996.

[W.F92] W.F.Opdyke. *A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[WS93] W.Li and S.M.Henry. Object-oriented metrics that predict maintainability. In *Journals of Systems and Software*, 1993.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005.

Vita

Suchitra S. Iyer was born in Mumbai, India on July 14, 1982. She obtained her Bachelor of Technology degree in Information Technology from Usha Mittal Institute of Technology, S.N.D.T. Women's University, Mumbai, India in June 2004. She worked with Hexaware Technologies Ltd., Navi Mumbai as a Software Engineer till June 2006. In Fall of 2007, she joined the Dept. of Electrical and Computer Engineering, University of Texas in Austin for pursuing her graduate studies.

Permanent Address: 16516 Castletroy Dr
Austin, TX 78717

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.