An Empirical Study of Package Coupling in Java Open-Source

A Thesis submitted for the degree of Doctor of Philosophy

by

Asma Mubarak

Department of Information Systems, Computing and Mathematics

Brunel University

2010

ABSTRACT

Excessive coupling between object-oriented classes in systems is generally acknowledged as harmful and is recognised as a maintenance problem that can result in a higher propensity for faults in systems and a ‘stored up’ future problem. Characterisation and understanding coupling at different levels of abstraction is therefore important for both the project manager and developer both of whom have a vested interest in software quality. In this Thesis, coupling trends are empirically investigated over multiple versions of seven Java open-source systems (OSS). The first investigation explores the trends in longitudinal changes to open-source systems given by six coupling metrics. Coupling trends are then explored from the perspective of: the relationship between removed classes and their coupling with other classes in the same package; the relationships between coupling and ‘warnings’ in packages and the time interval between versions in Java OSS; the relationship between some of these coupling metrics are also explored. Finally, the existence of an 80/20 rule for the coupling metrics is inspected. Results suggest that developer activity comprises a set of high and low periods (peak and trough’ effect) evident as a system evolves. Findings also demonstrate that addition of coupling may have beneficial effects on a system, particularly if they are added as new functionality through the package Java feature. The fan-in and fan-out coupling metrics reveal particular features and exhibited a wide range of traits in the classes depending on their high or low values; finally, we revealed that one metric (fan-in) is the only metric that appears consistently to exhibit an 80/20 (Pareto) relationship.

DECLARATIONS

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Asma Mubarak

date 30/04/10

DEDICATION

This thesis is dedicated with endless love and eternal respect to: my husband Issam, who has been always supportive with lots of love, patience, advice and jokes, my daughter Bayan, who brings happiness and cheerfulness to my life every time I look at her lovely face, and my parents and my sisters, who give me an unconditional love and support despite the distance.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Steve Counsell, for his advice, guidance, support and encouragement throughout my PhD work. Our discussions always gave me the confidence of being on the right way. I also owe special thanks to Professor Robert Hierons, my second supervisor, for his encouraging comments and suggestions throughout this research. It is a pleasure to thank all my colleagues in the Department of Information Systems, Computing and Mathematics at Brunel University and those outside.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CBO	Coupling Between Objects (Chidamber and Kemerer, 1994)
DIT	Depth of Inheritance Tree (Chidamber and Kemerer, 1994)
EXT	Number of EXTernal methods called (EXT) (JHawk, 2008)
FIN	Fan IN (JHawk, 2008)
FOUT	Fan OUT (JHawk, 2008)
LCOM	Lack of COhesion in the Methods of a class
LOC	Lines Of Code
MPC	Message Passing Coupling (Li and Henry, 1993)
MOOD	Metric for Object-Oriented Design
NOA	Number Of Attributes (Lorenz and Kidd, 1994)
NOC	Number Of Children
NOM	Number Of Methods (Lorenz and Kidd, 1994)
OO	Object-Oriented
OSS	Open-Source Systems
PACK	Number of imported PACKages
RFC	Response For a Class (Chidamber and Kemerer, 1994)
SE	Software Engineering
WMC	Weight Method per Class (Chidamber and Kemerer, 1994)

LIST OF PUBLICATIONS

1. Counsell, S., Mubarak, A. and Hierons, R. (2010) An evolutionary study of Fan-in and Fan-out metrics in OSS. *Proceedings of the 4th International Conference on Research Challenges in Information Science (RCIS 2010),* Nice, France, 2010.

2. Mubarak, A., Counsell, S. and Hierons, R. (2009) Does an 80:20 rule apply to Java coupling? *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering,* (EASE), Keele, UK.

3. Mubarak, A., Counsell, S. and Hierons, R. (2008a) An empirical study of "removed" classes in Java open-source, *Proceedings of the 4th International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 08).*

4. Mubarak, A., Counsell, S. and Hierons, R. (2008b) Empirical observations on coupling, code warnings and versions in Java open-source, *Proceedings of the3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2008),* Brno, Czech Republic.

5. Mubarak, A., Counsell, S., Hierons, R. and Hassoun, Y. (2007) Package evolvability and its relationship with refactoring, *Proceedings of the 3rd International ERCIM Symposium on Software Evolution*, Paris, France.

6. Mubarak, A. Counsell, S., Hierons, R. and Hassoun, Y. (2007) Package evolvability and its relationship with refactoring, *Electronic Communication of the European Association of Software Science and Technology*, 8.

CHAPTER 1. INTRODUCTION

1.1 Introduction

A software system is modified and developed many times throughout its lifetime to maintain its effectiveness. In general, it grows and changes to support the increasing demands in information technology. Consequently, the majority of software engineers today are concerned with changing and modifying existing software systems. In Software Engineering (SE), software maintenance is the process of making modifications to an existing system; software evolution is a term used to refer to the development of a system and its continuous change (Dvorak, 1994).

One of the most popular structures for building systems is object-orientation. In this approach, concepts of classes and packages are used. Each package contains a set of related classes, and packages are hierarchically organised in a package tree (Hautus, 2002).

From a maintainability perspective, *refactoring* plays a significant role in this field of software development activity (Fowler, 1999). Refactoring refers to a technique whereby changes are made to a program to improve its design without necessarily changing the semantics of the program (Fowler, 1999). As well as a better program design, the benefits of the refactoring include improvement program understandability and, in theory, improvement in the maintainability of that program. Fowler (1999) presents 72 types of refactorings with the motivations and the mechanics of each refactoring. There are numerous refactorings pertaining specifically to inheritance in the set of 72 refactorings of Fowler. For example, the ‘Extract Subclass’ refactoring creates a subclass for an existing class.

Software metrics have become essential in some disciplines of software engineering. They are used to measure software quality and to estimate the cost and effort of software projects (Fenton and Pfleeger, 2002). In the field of software evolution, metrics can be used for identifying stable or unstable parts of software systems, identifying where refactorings can be applied or have been applied, and detecting increases or decreases of quality in the structure of evolving software systems (Demeyer et al., 2000).

1.2 Motivation

Object-oriented (OO) design and development is very popular in today's software development environment. OO development requires not only a different approach to design and implementation than that of procedural but also a different approach to software metrics. Since OO technology uses objects and not procedures as its fundamental building blocks, the approach to software metrics for OO programs must be different from the standard metrics set. There have been many proposed OO metrics in the literature. As the quality of OO software is a complex concept, the aspects of the studied quality should be defined in order to decide how to measure them. Design metrics can be classified into two categories: static and dynamic (runtime). Static metrics measure what may happen when a program is executed and are said to quantify different aspects of the complexity of the static source code. Run-time metrics measure what actually happens when a program is executed. They evaluate the source code's run-time characteristics and behaviour. The metrics that are investigated in this Thesis are static coupling metrics. Stevens et al. (1974) first introduced coupling in the context of structured development techniques. It defined coupling as “the measure of strength of association established by a connection from one module to another”. It stated that the stronger the coupling between modules, that is, the more inter-related they are, the more difficult these modules are to be understood, changed and corrected and thus the more complex the resulting software system.

Excessive class coupling has often been related to the tendency for faults in software (Briand et al., 1997). A class that is highly coupled to many other classes is an ideal candidate for re-engineering or removal from the system to mitigate current and potential future problems. It is widely believed in the OO software engineering community that excessive coupling between classes creates a level of complexity that can complicate subsequent maintenance and potentially lead to the seeding of further faults (Briand et al., 1997). Moreover, a highly coupled class is expected to grow to be a relatively large class, making it even more appropriate, theoretically, to be removed from the system.

The purpose of the research in this Thesis is to investigate coupling metrics in the evolution of Java Open-Source Systems (OSS). In other words, the trends in and characteristics of coupling metrics and their changes as systems evolve are explored.

1.3 Thesis Objectives and Contribution

The main objective of this research is to assess how a system changes through the analysis of packages in the system and to compare that data with corresponding results from refactoring the same system. Another objective of the research is to explore the relationship between coupling metrics and the classes removed from multiple versions of several open-source systems; a further objective is to empirically explore coupling in these Java systems using coupling metrics, version release times and code warnings. Finally, we aim to explore whether an 80/20 rule exists in Java from coupling metrics over multiple versions of open-source software and to investigate the characteristics of classes with the highest values of incoming coupling metrics, notably FIN.

These objectives can be listed as follows:

1. To investigate versions of OSS with particular reference to the characteristics of classes removed from systems during their evolution. In particular, to conduct a thorough investigation of the removed classes from the perspective of their coupling to other classes, their size compared to other classes and their change trends before they were removed.
2. To discover the relationship between changes in coupling metrics over the releases of a system and the different time periods between these releases. While there have been many studies of evolving systems, the time frame between releases is often ignored and each version release is considered as occurring at an equal time interval from the last. Moreover, we aim to

investigate how extracted code warnings could help in understanding the patterns of maintenance activity in which increased coupling will inevitably feature.

3. To explore whether an 80/20 rule exists in Java coupling metrics over multiple versions of OSS. This will help in identifying the 'key' classes, defined as certain classes in any system that comprise a large number of methods and, by implication, a large amount of coupling.
4. To investigate the characteristics of classes that shown the highest value of incoming coupling metrics. A class that is highly coupled to many other classes is an ideal candidate for re-engineering or removal from the system to lessen both current and potential future problems. A problem that immediately arises, however, for the developer when considering re-engineering of classes with high coupling is the size of the dependencies of those classes and the kind of dependencies, 'incoming' or 'outgoing'. We also address the issue of potential re-engineering and view coupling as a key contributor to the decision on whether and when to re-engineer (classes) or not over the lifetime of a system.

This Thesis makes a number of contributions from an empirical perspective; in particular, from an evolutionary perspective. It informs the empirical understanding of coupling features and the contributions have been published in various archived sources. The contribution of the research in this Thesis can also be demonstrated on the basis that previous researchers (Kemerer and Slaughter, 1999a; 1999b) have postulated that software evolution has not been the subject of significant research. Consequently, they expressed the need for further empirical studies of software evolution.

The main contribution of the Thesis is first how changes in the maintenance practice may help a project manager to approximate the potential maintenance effort needed for the system, and for the project's developers to take preventive action in the form of additional system maintenance and refactoring. Secondly, since few empirical studies have analysed coupling from an evolutionary respective, we believe the results in this Thesis form a contribution to our understanding of how coupling evolves and where the majority of maintenance changes are applied. Finally, all the data used in this Thesis is available to other researchers for the purposes of replication and, in this sense, we see the Thesis as a contribution to the ongoing body of empirical research in this area.

1.4 Preliminaries

Seven OSS were chosen to conduct the research investigation. These systems were all written in Java with sufficient versions to allow a meaningful longitudinal analysis. Systems were selected in terms of 'number of downloads' order from sourceforge. The selection process thus resulted in many systems being rejected from candidate systems identified because they were either a mix of different languages and/or did not contain multiple versions for download. These systems are Velocity, Jasmin, SmallSQL, pBeans, Asterisk, DjVu and JWNL. More details on these systems are available in Chapter 3.

Software metrics (Fenton and Pfleeger, 2002; Chidamber and Kemerer, 1994; Lorenz and Kidd, 1994) are a significant part of our investigation. In this Thesis, we make use of software metrics as the basis of our analysis to explore quantitatively the changes of coupling in multiple versions of the studied systems. The following six independent coupling metrics were collected using JHawk.

1. Response for a Class (RFC).
2. Number of EXTernal methods called (EXT).
3. Message Passing Coupling (MPC).
4. PACK. Number of imported PACKages.
5. Fan-in (FIN).
6. Fan-out (FOUT).

We also collected the total number of methods (NOM) and the lines of code (LOC) in each class as size measures. Again, these metrics will be described in more detail in the methodology chapter (Chapter 3).

We also collected a physical time-based metric, which is the actual time interval between each version release.

1.5 Thesis Outline

The thesis is structured over eight chapters. We next explain the contents of each chapter and how the chapters in totality knit together to form a coherent research story. This chapter presents the context and motivation of the work, and gives the overview of the objectives and contributions.

Chapter 2 describes related work to the research problems addressed. It looks at related and complementary work in the area of OSS, coupling metrics and refactoring. It also provides insights and justification for the nature of the research presented in this Thesis.

Chapter 3 provides a detailed description of the research methodology adopted in the Thesis including an explanation for the basis upon which the systems used in the study were chosen, description of the software metrics used in the research, and justification for the choice of statistical analysis used.

Central to the aim of the Thesis is to uncover traits in OSS from an evolutionary perspective (at the package level). Chapter 4 assesses how a system changes through the analysis of the said packages in a system and compares the obtained data with corresponding results from refactoring the same system. Knowledge of trends and changes within packages is a starting point for an understanding of how effective the original design may have been and how susceptible types of packages may change. It can also inform our knowledge of facets of software such as coupling and cohesion.

One aspect of evolution detailed in Chapter 4 and a key observation was the dynamic nature of systems and, in particular, the tendency for removal of classes as a system evolved. Chapter 5 investigates versions of OSS with particular reference to classes 'removed' during their evolution. The research explores whether classes removed from the system are lowly or highly coupled relative to other classes in the same package. Moreover, it explores the size of the same classes if they are excessively large compared with the remaining classes in the package. Finally, the changes of the removed classes before they are removed are assessed to identify patterns of change.

In Chapters 4 and 5, a key assumption made was that evolution and the versions of each system occurred at equal intervals in time. This assumption could be criticised on the basis that frenetic change activity could easily occur at irregular intervals. Chapter 6 therefore investigates the trends in change activity that can be observed if we factor in the different time periods between releases of a system. Hence, in this chapter the relationship between coupling and the potential code warnings (i.e., areas of code that might prove problematic) is investigated. The FindBugs tool was used to highlight potential sources of code problems.

One observation made from the studies in Chapters 4, 5 and 6 was that the bulk of changes and coupling activity (identified by the metrics collected) centred around a small number of classes, while the vast majority of classes remained untouched throughout the same versions studied. Pareto's Law or an 80/20 rule is a naturally occurring phenomenon which suggests that 80% of class activity occurs in just 20% of classes. Chapter 7 therefore explores, first, whether an 80/20 rule exists in Java from six coupling metrics over multiple versions of open-source software and, if so, whether that relationship is exacerbated over time. After that, the characteristics of classes revealing the highest fan-in are investigated. Finally, the trends of the changes in the fan-in and fan-out are inspected in addition to the relationship between these two metrics.

Chapter 8 provides the conclusions and the contributions of the research presented in this Thesis with reflection on the original objectives and the level to which they were achieved. It also gives some thoughts about related future research.

CHAPTER 2. LITERATURE REVIEW

2.1 Overview

In the previous chapter, we gave an introduction to the Thesis and we presented its structure. In this chapter, we describe related work to the research carried out in this Thesis. First, some concepts are defined, such as *empirical software engineering*, *software life-cycle* and *software maintenance and evolution*. After that, issues in evolving a system in terms of class and package changes are stated. Finally, related work to the areas of software metrics, coupling metrics and refactoring in particular are described.

In Section 2.2, we talk about the idea of empirical work in software engineering. Section 2.3 presents a description of related research in software maintenance and evolution. Some concepts in the OO paradigm are described in Section 2.4 in terms of classes and packages. In Section 2.5, we review the software metrics presented in the literature, and how they have been used in practice. Section 2.6 provides a detailed analysis of published work on OO coupling. Finally, we provide an analysis of published work on software refactoring in Section 2.7. A summary of the chapter is presented in Section 2.8.

2.2 Empirical Software Engineering

According to McDermid (1991, cited in Bennett, 1996), Software Engineering SE can be defined as “the science and art of specifying, designing, implementing and evolving - with economy, time limits and elegance - programs, documentation and operating procedure whereby computers can be made useful to man”. This definition of software engineering is complete and contains the essence of these concepts. It declares that it is a science, and thus clarifies that it is about the task of looking for knowledge and scientifically managing that knowledge. It points to art to indicate creativity. It presents four actions, which inform the real work carried out. The expression ‘economy’ suggests that in some way management has to be involved; ‘time limits and elegance’ indicate that an organised and methodical approach is significant. The outcome artefacts are specified to be the program, documentation and the operating procedures. Finally, ‘useful to man’ underlines the significance of never neglecting the essential purpose - the human being.

Software Engineering is still a very young branch of computer science (Bennett, 1996). It emerged because of a necessity for new notations, new methods and new tools that could respond to the raised complexity of development and software systems. Moreover, it can be said that software engineering contains theories, techniques, methods and tools required to develop reliable software. Because of all these, the need for an empirical approach arose. Empirical Software Engineering focuses on the evaluation of software engineering technologies. It attempts to assess models and techniques, and to investigate how they perform in practical frameworks, with the aim of creating a database to support decision making for the progress (Basili et al., 1996a). A set of hypotheses are formulated to declare an assumption on how relevant variables are influenced by other independent variables. After that, these hypotheses are validated by conducting an experimentation process. Usually, this is decided by a statistical analysis conducted on the collected data.

In 1996, Wasserman stated that software engineering had eight technical characteristics including the software life-cycle. A software life-cycle is defined as the period of time which begins when a software product is designed and finishes when the software is not used anymore (Longstreet, 1990). The software system goes through several phases throughout its life-cycle. According to Pillai (1996), these phases can be divided into the following stages:

- Requirements definition and analysis phase, distinguished by exploration and analysis of the description of the product.
- Design phase, in which we design drafts and test their integrity.
- Implementation and testing phase, when all test cases are executed.

- Installation phase, which determines that the system is ready to be released to customers.
- Maintenance phase, which includes regression testing.

The most costly phase is the maintenance phase, because of the amount of change that occurs in the system (Williams and Carver, 2007). It costs between 40% and 90% of the total life-cycle costs; however, it was not recognised as a serious activity until 1970s (Bennett, 1996). Kajko-Mattsson et al. (2001) state that although software maintenance forms a main phase of the software lifecycle; it has frequently been ignored and is given very little consideration in both educational and manufacturing fields.

2.3 Software Maintenance and Evolution

The software maintenance phase comes after the implementation of a system. The maintenance stage should be initiated for various reasons. Burd and Munro (2000) state that these reasons could be sorted into four categories of maintenance activities, which are:

- Perfective maintenance: implies enhancing the functionality of software in reply to a user's identified changes.
- Corrective maintenance: entails the correction of errors that have been defined in the software.
- Adaptive maintenance: involves the alteration of the software that is caused by changes within the software situation.
- Preventative maintenance: implies updating the software to progress upon its future maintainability without changing its existing functionality.

One of the most important issues presently facing software engineering is the capability to evolve a system with the changing requirements of its stakeholders. When systems evolve, many issues arise. According to Perry (1994), the dimensions of the context in which the system evolves help to recognise the evolution of software system appropriately. He characterises these dimensions as: the fields that are related to these system, the skills learnt from evolving and employing these systems and the procedures used in manufacturing and evolving these systems.

Zimmermann et al. (2005) try to lead programmers along associated modifications by relating data mining to the histories of versions. To this end, they use the ROSE tool which aims: firstly to give a programmer recommendations and forecast of probable changes by using data mining skills to get these related changes. Secondly, to distinguish coupling among program items that program analysis cannot distinguish. Lastly, the tool will warn the program if the changes that the user wants to perform are incomplete.

In terms of software evolution, the laws of Lehman (Belady and Lehman, 1976) provide the backdrop for many past evolutionary studies. Evolution has also been the subject of simulation studies (Smith et al., 2006) and this has allowed open-source software evolution to be studied in a contrasting way to that empirically.

One of the main issues that arise when systems evolve is which patterns of change apply at different levels of abstraction. By studying how classes change we can determine how the system changes. Developers can make changes to a built system by producing new classes instead of modifying existing ones (Bieman et al., 2003). Bieman et al. found that there was a relationship between design structures and development and maintenance changes. They examined whether potential changes to a class could be predicted by the architectural design context of a class, and found that a correlation between class size and number of changes was inconclusive. Moreover, they found that in four of five case studies, classes which had function in design patterns were modified more frequently than other classes (Gamma et al., 1995).

In another study carried out by Bieman, Jain and Yang (2001) it was found that maintenance effort could be affected by certain design factors. For example, it was found that there was a correlation between class size and the number of changes. Moreover, two unexpected relationships were discovered. The first related to the classes reused during inheritance; it was found that these classes tend to be changed more. The second relationship identified was that classes recognised as prone to change were the classes which played a part in design patterns.

2.4 Concepts in OO Software

Systems are built using many different structures. One of the most popular structures is OO. In this approach, a computer program may be considered as a collected composition of separated units, or objects, each one able to receive messages, process data, and send messages to other objects. Bennett et al. (2002) state that OO is organised around the interfaces of the objects, their status at a particular instance and how the objects communicate with each other. Moreover, they define the class as a set of related operations and attributes that defines a class's behaviour, methods and attributes. Consequently, an object is an instance of class and has identity, behaviour, and state. In other words, they declare that the purpose of a class is to state a group of methods, operations and attributes that completely illustrate the behaviour and structure of objects. Each object consists of a single identifier, a set of attributes and a set of methods. Each attribute has a value that can change, and an object's method is invoked as a reaction to a message from another object (Jajodia and Kogan, 1990).

Systems which are built using an OO structure are potentially more difficult to maintain than those in the procedural structure as the existence of inheritance and polymorphism raises dependencies in a program and incorporate potential difficulties in program understanding and analysis (Wilde and Huitt, 1992). Wilde and Huitt summarise the most important problems that can be expected in maintaining OO programming as follows:

- The problem of dynamic binding: dynamic binding gives much of the flexibility of OO languages; however, it may cause problems in outlining dependencies within the program, which many maintenance tools depend on.
- Dependencies in an OO system: a dependency in a software system can be considered as a straight connection between two entities in the system; any

change in the first entity may affect the second. In an OO system, using polymorphism and hierarchy produces an increase in the types of dependencies: class-to-class dependencies, class-to-method, class-to-message, class-to-variable, method-to-variable, method-to-message and method-to-method. Because of the multidimensional nature of the connections, it is very hard to sift through all the relationships in a system, and that makes program understanding more difficult.

- The structure of an OO program: an OO system may contain a number of very small modules as for many tasks very short methods may be written. Consequently, the code for any particular task would be very broadly distributed. Understanding a line of code might need an understanding of a series of method invocations through some distinct object classes to discover where the task is actually completed.
- High-level system understanding: when a maintainer wants to become familiar with a system for the first time, high level system understanding is required. In order to provide this to the maintainer, statistical clustering tools are needed to structure an OO environment.
- Locating system functionality: according to the dispersion of functionality into different object classes in OO system, there is some complexity in discovering where different functions are performed. Therefore, maintainers may need to use tools to investigate and compare traces of system performance in order to help them recognise the methods and classes related in a functional sense.
- Detailed code understanding: maintainers spend long periods of time understanding the detailed code that they intend to modify. In an OO environment, detailed code understanding is very complicated because of the class hierarchy and associated features such as polymorphism. By using the concept of dependency analysis, maintainers can easily identify the compound types of dependencies in OO programs. They also identify chains of relationships, which may be helpful in tracing through widely distributed code fragments.

Classes in Java are organised into separate groups called packages. The aim of a package is to join strongly related classes within a single entity and to offer confidential access between those classes. Each package contains a set of related interfaces, classes and exceptions, and packages are hierarchically organised into a package tree (Hautus, 2002). At the present time, most programs are built in terms of a set of classes, or packages, and this is enhanced by the appearance of the concept of encapsulation. Bennett et al. (2002) state that encapsulation refers to the capability for the same message to be sent to objects in different classes, each of which replies to the message in a different way. Consequently, the full idea of encapsulation in OO programming is for the methods and variables of objects to be protected against unauthorised access by other objects. That can be achieved through the access modifiers, which Bennett et al. classify into four different types:

- Public: a feature (an attributes or a method) with a public access can be accessed by any object.
- Private: a feature with a private access can be accessed only by an object from the class that includes in it.
- Protected: a feature with a protected access can be accessed either by an object of the class includes in it or of a subclass or descendant of that class.
- Package: a feature with a package access is accessed only by an object from a class in the same package.

In the environment of OO applications (according to Ducasse et al. (2005)), packages have varying functions: they may include utility classes used through their structure, or they may include some fundamental subclasses enlarging a framework. In addition, they indicate that as classes are included in packages, it is essential in the re-engineering and development of OO systems to understand sets of classes and packages; they add that packages are more than a simple generalisation of classes. Depending on the relationship between packages and their contained classes, we can decompose the application and re-modularise it. Ducasse et al. (2005) intend to help reengineers and researchers working on remodularisation to gain a better understanding of OO programs. They give two radar visualisations called ‘butterfly views’ which assist in comprehending and classifying packages. These butterfly views represent how a package connects to the remaining parts of the system and they also illustrate how a package is internally structured.

In another study, Ducasse et al. (2004) state that understanding packages is an essential action in the re-engineering of OO programs, and the cost of modifying the program may be influenced by the correlation between packages and their enclosed classes. In order to support the developer in achieving a mental image of an OO system and understanding its packages, Ducasse et al. (2004) introduce a top-down engineering method based on visualisation. Consequently, they raise the abstraction level by detecting packages rather than classes. They classify packages by supplying a polymetric observation that helps the engineer to concentrate on packages rather than being flooded with information. They also illustrate how a package communicates with the remaining parts of the program and give an idea about how a package is built internally.

Hautus (2002) observes that many researchers try to comprehend programs by considering the analysis and visualisation of them. However, packages are essential as they are well-suited for identifying the sophisticated design of Java programs. Therefore, focus should be on packages rather than classes or methods in research, they also present the Package Structure Analysis Tool (PASTA). The PASTA metric is described as: “the weight of the undesirable dependencies between the sub-packages divided by the total weight of the dependencies between the sub-packages”. Hautus states that this metric gives a means of speedily estimating the inner value of complex software products based on their source code.

2.5 Software Metrics

In this Thesis, we use software metrics to empirically investigate the trends of coupling in the evolution of Java OSS. Generally speaking, software metrics are used to explain the activities concerned with measurement in software engineering, and to offer information to support decision making during software development and testing from a technical and managerial side. The metrics which we use enable us to measure coupling in each version of the systems under study and to represent the evolutionary behaviour of systems from a coupling viewpoint, which helps to inform software quality and software resource requirements.

Software metrics vary from generating numbers, which characterise properties of software code through, to models which help predict software quality and software resource needs. They are used to measure attributes of software systems as well as recognise the software threats and decrease the cost of developing and maintaining the software by taking corrective action early in the development course (Hall et al., 2005).

Measurement, according to Fenton and Pfleeger (2002), is a mapping of empirical objects to statistical objects with consideration given to all structures and relationships. The attributes measured by software metrics can be categorised into two groups: internal and external attributes. The internal attributes of a software system include size, coupling and the amount of reuse in the system, while the external attributes include usability, reliability and security of a system (Fenton and Pfleeger, 2002). In this Thesis, since we are concerned with coupling in the system, we measure the internal attributes of the studied systems. There is also a distinction to be made between *direct* and *indirect* measurement of attributes (Fenton and Pfleeger, 2002). Direct measurement of an attribute of an object involves no other attribute or object. For example, the length of source code is measured by lines of code, and the number of defects discovered during the testing process is measured by actually counting the defects. Indirect measurement of an attribute of an object involves other attributes or objects. Examples of the indirect measures are: the module defect density which is the number of defects divided by the module size, and the requirement stability, which is the number of initial requirements divided by the total number of requirements. In our Thesis, we measure direct and indirect attributes.

The first text on software metrics was published in 1976 by Tim Gilb (1967). Therein, Lines of Code (LOC) was used to measure program quality and productivity (Fenton and Neil, 1998). Akiyama (1971) proposes a basic regression model for module density to measure program complexity, and that represented the first step in using metrics for predicting software quality.

By the introduction of OO languages, the main feature of academic research has been to refine, extend and validate complexity metrics (Chidamber and Kemerer, 1994; Lorenz and Kidd, 1994; Abreu and Carapuca, 1994; Briand et al., 1998; Briand et al., 1999b; Harrison et al., 1998; Arisholm et al., 2004).

Chidamber and Kemerer (C&K) (1994) proposed six OO metrics as a suite to measure features of OO systems. The suite of metrics consisted of Weighted Method per Class (WMC): measures the number of methods defined in a class, Response For a Class (RFC): measures the total number of methods that can be executed as a result of receiving a message from an object of that class, Lack of Cohesion in Method (LCOM): measures the lack of cohesion in methods of a class, Depth of Inheritance Tree (DIT): measures the maximum number of classes from a leaf to the root class in an inheritance hierarchy, Number Of Children (NOC): measures the total number of descendent classes from a single class, and Coupling Between Objects (CBO): measures inter-relationship of classes.

These metrics have been used extensively since in a variety of studies; none of the metrics, however, give a coarse-grained feel for the incoming and outgoing coupling that OO fan-in and fan-out provide and which are the subject of some work in this Thesis. Of these six metrics, the RFC seems to have been the least favoured by empirical software engineers and yet there is no obvious reason for neglect of its investigation (our study therefore attempts to redress that imbalance). The majority of empirical studies in OO seem to have focused on other C&K metrics such as the DIT, WMC and CBO. In our study, we focus on the RFC and CBO metrics. C&K metrics appear to be useful for developers and designers of systems as they operate at the class level (Basili et al., 1996b).

Abreu and Carapuca (1994) identify the MOOD (Metrics for Object-Oriented Design) set of metrics which fulfil some evaluation criteria. These criteria include the requirements for formal definition for metrics determination. Moreover, the metrics should be obtainable early in the system life-cycle, language independent, dimensionless, down-scaleable and easily calculated. The MOOD set of metrics comprised 1) Method Inheritance Factor (MIF): The ratio of the sum of the inherited methods to the total number of available methods; Attribute Inheritance Factor (AIF): The ratio of the sum of the inherited attributes to the total number of available attributes; Coupling Factor (CF): This metric considers the actual couplings among classes in relation to the maximum number of possible couplings; Polymorphism Factor (PF): calculates the degree of method overriding in the class inheritance tree; Method Hiding Factor (MHF): The ratio of the sum of all the hidden methods to the total number of methods; and Attribute Hiding Factor (AHF): The ratio of the sum of all the hidden attributes to the total number of attributes. These metrics help in setting the OO design measures at the organisation level and help OO practitioners to conduct their development processes.

Lorenz and Kidd (1994) introduce a set of metrics to measure dimensions of OO systems. Most of these metrics are direct metrics and include directly countable measures. The metrics are divided into four categories: size, inheritance, internals and externals. They include Number of Methods per class (NM), Number of Public Methods per class (NPM), Number of Variables per class (NV), Number of Public Variables per class (NPV), Number of Methods Inherited by a subclass (NMI), Number of Methods Overridden by a subclass (NMO), Number of Methods Added by a subclass (NMA), Average Methods Size (AMS), Number of times a Class is Reused (NCR), and Number of Friends of class (NF).

According to Shepperd (1995), theoretical and empirical validations are essential for the success of the software metrics when using them in practice. Metrics validation is the procedure of investigating if the software metric precisely measures the software attribute which they purport to measure (Fenton and Pfleeger, 2002).

Harrison et al. (1998) investigated a set of six OO software metrics, called the MOOD metrics, with measurement theory perspective and taking into consideration the OO features that they were meant to measure: encapsulation, inheritance, coupling and polymorphism. By applying these metrics to empirically investigate three different application domains, they found that the MOOD metrics set could be used to provide a general assessment for the systems studied. Their results showed that MOOD metrics work at the system level and hence are useful for project managers.

Briand et al. (1999a) introduced an outline of the existing empirical studies of OO systems, methods, tools, notations and processes and discussed four directions for further work in the area of empirical OO software development and evolution. These directions were: categorise main quality and productivity issues, assess and compare OO technologies, construct (productivity and quality) models and meta-analysis. They highlighted points to be considered to accomplish successful empirical studies. In particular, they encouraged cooperation with the software industry in an attempt to improve the quality and productivity of empirical studies.

Validation is important to the success of software measurement (Kitchenham et al., 1995a). Kitchenham et al. propose a validation framework to demonstrate how software metrics should be investigated, to help practitioners and researchers to figure out how to validate a metric, and to identify when it is suitable for a metric to be applied. They differentiate between two fundamental assessment methods: *theoretical validation*, which validates that the measurement obeys the measured element's essential properties and *empirical validation*, which confirms that values that measure attributes are consistent with values expected by models involving the attribute. Moreover, Kitchenham et al. define a set of criteria that all measures must obey to be determined a valid measure from a measurement theoretic viewpoint.

2.6 Coupling Metrics

In 1974, Stevens et al. first defined coupling in the context of structured development as “the measure of the strength of association established by a connection from one module to another” (Stevens et al., 1974). Coupling metrics are OO metrics that measure the interdependence between a given class and the other classes in the system. Classes are coupled when methods in one class use methods or attributes of other classes.

A large number of researchers have tried to understand how to assess the quality of OO design. However, external quality measures such as maintainability and reliability cannot be measured until late in the software life cycle. Therefore, there is a need to recognise early predictors for such qualities. A number of research studies have used static coupling metrics to measure the maintainability of OO systems (Arisholm et al., 2004). Those measures have been helpful predictors of several attributes like modifications and fault-proneness. For example, Arisholm et al. (2004) describe the use of dynamic coupling metrics. These metrics are major signs of change-proneness and they go together with static coupling metrics capturing different facets of a system. There is also some evidence that some forms of coupling have a negative impact on fault-proneness (Briand et al., 1997). As a violation of encapsulation, C++ friends - an improper form of coupling, have also been shown to reflect higher fault rates in software (Briand et al., 1997). Briand et al. defined coupling measures and empirically found that several export and import coupling measures were significant predictors of fault-proneness. Additionally, they found that using “friend” classes in C++ increased the fault-proneness of classes more than other kinds of coupling metrics. English et al. (2007) presented metrics that were refinements of the work of Briand et al. (1997), and assessed these metrics using the LEDA software system. They found that the metrics depended on the ‘friend’ type of coupling (applied frequently to access hidden attributes in classes, but rarely to access hidden methods). They differentiated between coupling metrics that used the ‘friend’ mechanism, ‘inheritance’, and ‘other’ forms of coupling. They further stated that metrics that depended on ‘friends’ and ‘other’ forms of coupling were different to existing metrics, and were helpful in both prediction models and conducting a more thorough investigation of the structure of software systems.

Briand et al. (1999b) carried out a widespread study of the currently available coupling literature in OO systems and introduced a framework for the definition, comparison and assessment of coupling measures in OO systems. The framework consisted of six criteria, which were important in identifying a coupling measure. The six criteria were: *locus of impact, type of connection, stability of server, granularity of the measure, direct or indirect connections and inheritance.* They concluded that even though many studies have been carried out, there are too few empirical studies of (coupling) measurement, particularly OO. This leads to a delay in research exploring suitable solutions for practitioners. .

Bartsch and Harrison (2006a) extend Briand's framework for AspectJ concentrating on a specific definition of different coupling connections found in AspectJ (2005). The criteria of the framework, which have to be thought about when designing, analysing and comparing OO measures, are seven. Six of these criteria are those introduced in the Briand et al. framework, and the seventh is *Instantiation,* which refers to whether or not to count aspects at a per-instance level (Bartsch and Harrison, 2006a). Bartsch and Harrison (Bartsch and Harrison, 2006b) use these criteria again in another paper to evaluate five coupling metrics proposed by Ceccato and Tonella (2004). They found that none of the coupling measures could be validated in the context of the validation framework used; however, most of them do not show any key problems and the quality of most of them can be increased by more accuracy in their definitions (Bartsch and Harrison, 2006b).

Li and Henry (1993) support the view that excessive coupling makes maintenance and tracing more difficult. In their research, they focused on ten OO software metrics and then validated these metrics with maintenance effort on two OO systems. They found that maintenance effort was related strongly to the metrics and it could be predicted from the combinations of these metrics. Moreover, they proved that this prediction was successfully cross validated.

The role of method invocation (a form of coupling between classes) in creating faults is also highlighted by the work of Briand et al. (1998). In this work, they tried to validate all the OO measures found in the literature, especially the impact of these metrics on class fault-proneness, and their ability to predict fault locations. Results have shown that the possibility of identifying fault in a class is strongly related to many coupling and inheritance measures. The most important quality factors in creating faults are method invocations, depth of a class in its inheritance hierarchy and the scale of change in a class as a result of specialisation (Briand et al., 1998).

The Fan-in and Fan-out metrics of Henry and Kafura (1981) measure the number of inputs and outputs of a given module, respectively. According to Henry and Kafura (1981), the information flow between system components is a practical and appropriate technique for measuring large-scale systems since this technique exposes more of the system connections than other ordering relations, and the main elements in this technique can be concluded at design phase. In order to present their measurement for this technique, they defined fan-in and fan-out as follows. The fan-in of a module is the number of inputs plus the number of data structures from which the module gets information. The fan-out of a module is the number of outputs plus the number of data structures which the module updates (Henry and Kafura, 1981).

2.7 Software Refactoring

Refactoring is one of the techniques widely used to improve the structure of software systems. This technique was first introduced by Opdyke and Johnson (1990), referring to the internal structure development of an OO software system without changing the external performance of the system. Before that, Chikofsky et al. (1990) introduced the term software restructuring, which could be considered as the starting point of refactoring. They defined software restructuring as “the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behaviour (functionality and semantics)”. The research of Johnson and Foote (1988) and of Foote and Opdyke (1995) have all made considerable contributions to the refactoring discipline and also helped to reveal the viability and potential of refactoring. Refactoring is used to improve the design of the program and make it easier to understand (Counsell et al., 2006). Consequently, this supports software maintenance and reuse (Fowler, 1999; Johnson and Foote, 1988; Chikofsky et al., 1990). The link between maintenance as part of every system’s evolution and that dedicated to refactoring is a topical area for OO researchers and practitioners (Mens and Tourwe, 2004).

Refactoring was introduced in a seminal text by Fowler (1999). Fowler defines software refactoring as: “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour”. Moreover, Fowler describes the procedure of seventy-two different refactorings in four major categories and explains assorted ‘bad smells’ in code. Fowler (1999) gives a list of refactorings that can be useful for developers to improve the design of their code. Some representative categories of refactorings are: *Composing Methods, Moving Features between Objects, Simplifying Conditionals, Making Method Calls Simpler, Generalization, and Big Refactorings.* According to Fowler, the basic indicator of when refactoring is overdue is when the code begins to ‘smell’. Another approach has been demonstrated by Tourwe and Mens (2003), who use the technique of logic meta programming to detect bad smells and get the needed information for the proposed refactorings. By this, they show how support can be supplied for discovering when a design should be refactored and identifying which refactorings might be applied to develop this design.

A full survey of recent refactoring work can be found (Mens and Tourwe, 2004). These researchers provide an outline of the existing research being completed in software refactoring and restructuring. They consider refactoring activities such as: identifying where to refactor software, determining which refactorings to apply, making sure that the applied refactoring does not change behaviour, applying the refactorings and finally preserving the stability between the new code and other software artefacts. Mens and Tourwe then talk about different refactoring techniques including graph transformations and invariant, pre and post-conditions. They discuss refactorings related to the kinds of software artefacts, and end with a look at various tools that present support for automation, reliability, configurability, coverage and scalability.

We can improve the quality of design and reduce the complexity and the cost in succeeding development phases by applying refactoring as early as possible during the software life-cycle (Zhang et al., 2005). Moreover, developers should refactor 'mercilessly' and consistently (Beck, 1999). To identify places that need refactorings, developers should use software metrics before a refactoring to measure the quality of a software system (Mens and van Deursen, 2003). The metrics can also be applied after refactoring to measure the improvement in system quality. Programs that are not written in an OO language are harder to restructure because data flow and control flow are strongly interconnected (Mens and Tourwe, 2004). Nevertheless, refactoring of programs written in OO languages is not easy, particularly when we take into account complex OO facets such as inheritance, polymorphism or dynamic binding. For example, recent empirical work by Najjar et al. (2003) has shown that refactoring can give both qualitative and quantitative benefits – the refactoring 'replacing constructors with factory methods' of Kerievsky (2002) was used as a source.

Demeyer et al. (2000) detected refactoring indicators when comparing different versions of a software system. They used four heuristics to find refactorings, where each heuristic was identified as a combination of change metrics. The refactorings in the first heuristic split functionality from a class into a superclass, or combined a superclass with one or more of its subclasses. The second heuristic split functionality from a class into a subclass, or combined a subclass with one or more of its subclasses. The third heuristic explored the refactorings that moved functionality from one class to another, while the final heuristic explored the refactorings that split methods into one or more methods defined in the same class. In terms of investigating the link between refactoring and testing, Counsell et al. (2006) adapted a testing taxonomy suggested by van Deursen and Moonen (2002) built on the refactoring impact on the ability to use the same set of tests *'post-refactoring'*. They urged that when making refactoring decisions, there was a requirement to consider the inter-relatedness of refactorings.

In an empirical study of multiple versions of seven open source Java systems, Advani et al. (2005) explored the refactoring trends across these systems. They declared that simple refactorings, at the method and field level but not as part of larger structural changes to the code, were most commonly undertaken by developers, with no pattern across the different versions of the systems. However, refactorings predominantly occur in the middle versions of a system not in earliest and/or latest versions. Advani et al. (2006) also describe a tool for collecting refactoring data from multiple versions of Java systems. The tool was designed to extract refactoring information from Java systems. It collected information about fifteen refactorings from seven systems and compared this information for the different releases. They found that the tool was a good indicator for the major kinds of refactorings used by developers. We used this tool in the experiment described in Chapter 4 to investigate the cross-comparisons between the high-level package trends and refactoring practice, and to provide insights into why refactoring might be applied after a burst of regular change activity rather than consistently (Mubarak et al., 2007).

2.8 Summary

The central theme of this research is to demonstrate how evolving systems change during the transaction from one version to another in terms of coupling metrics. In this chapter, concepts related to this theme have been presented. We included definitions of issues linked to software evolution regarding changes in systems from the sense of the changes in the contained classes and packages. In the next chapter, we provide an explanation of the research process approach and the rationale for the research methods selected in this Thesis. A case study strategy will be described in detail. Finally, the systems under study will be explained accompanied by the study aims and objectives.

CHAPTER 3. RESEARCH METHODS

3.1 Overview

In Chapter 2, related work carried out in this Thesis was reviewed and analysed. This chapter aims to illustrate the research approach used to investigate our research. The chapter starts with an exposition of research paradigms and methods in the context of software engineering. Then it proceeds by providing a detailed explanation of the research objectives that guided the investigation and the rationale for the research methods selected. The research process includes a description of the systems under study, definition of the investigated metrics, explanation of the data analysis for this study and statistical methods employed.

Section 3.2 gives the description and justification for the research methods and strategies used in this Thesis. Details of the research objectives and hypotheses are presented in Section 3.3. Finally, a description of the research preliminaries is provided in Section 3.4.

3.2 Research Paradigms and Methods

A software engineer uses certain methods to estimate the existing work in order to raise the quality of a software product or reduce the cost of product improvement (Sommerville, 1996). By analysing data, certain conclusions can be used to predict how efficient and valuable work will be in the future in order to improve software quality. This data could be collected by the researchers using one of the data collection strategies such as surveys, questionnaires, interviews, experiments and project artefacts. Collected data can be quantitative or qualitative depending on issues such as personal experience and the nature of the research problems and questions. Creswell (2003) defines three approaches to research. These three approaches are quantitative, qualitative and mixed methods, and they are defined as follows:

- A quantitative approach: this approach is concerned with measuring a relationship or comparing two or more sets. It often uses an experiment, or data collected through a case study and helps in assessing the causes of a treatment. Quantitative data usually promotes statistical analysis.

- A qualitative approach: in this approach, human and social problems are studied and interpreted depending on explanations that people provide. This means that qualitative researchers study things in their natural settings, trying to make sense of experiences in terms of the justifications people bring to them. The focus in this approach is on developing theory and generating knowledge. The data is obtained from interviews, case studies and observations.

- A mixed method approach: in this approach knowledge claims are based on a practical basis and tend to combine or mix both qualitative and quantitative approaches.

The research conducted for this Thesis is quantitative in nature; it uses software metrics collected from several Java OSSs and related to coupling.

In general, for any research design there is a need to formulate a framework for a research design. Robson (2002) categorises these components as purpose, theory, research questions, methods and sampling strategy. Both the purposes which the study tries to achieve and the theory guide and inform the study and help to identify the research question. The methods and the strategies used determine the answer to these research questions. The strategies which can be used in the empirical investigations are varied; however, Robson (2002) defined three major different types of strategies that may be adapted:

- Experiment: an experiment is an instance of fixed research design. It is a particularly focused study and is usually done in a laboratory in a controlled environment. In this approach, one or more variables called *independent variables* are manipulated and the effects of this manipulation on one or more other variables, called *dependent variables*, are measured. All other variables are controlled.

- Survey: surveys are generally carried out as part of non-experimental fixed design. They aim to investigate areas by asking a broad collection of open-ended questions. They are normally carried out for descriptive reasons as they can supply facts about the distribution of a large range of features and of association between such features. However, surveys take a long time to be analysed, and they may not be an effective procedure. Moreover, surveys are only applied to a sample that represents the population studied.

- Case study: case studies are "a strategy for doing research which involves an empirical investigation of a particular contemporary phenomenon within its real life context using multiple sources of evidence" (Robson, 2002, p.178). A case study is a well-established strategy which focuses deeply on a process, a program, an event, an activity, or one or more individuals. Moreover, a case study takes place at particular times with particular people in particular places. It can be considered as an observational study as the control in it is low.

According to the design strategy, it can then be decided if the approach should be quantitative or qualitative. Wohlin et al. (2000) state that as experiments focus on evaluating various variables before and after making changes to them, they are merely quantitative. On the other hand, the same authors state that the categorisation of a survey or a case study relies on the collected data and the applied statistical methods held in a qualitative or quantitative approach. In a case study, data is collected for a particular reason during the study, and based on this data statistical analyses can be completed (Wohlin et al., 2000). Moreover, Wohlin et al. state that although case studies are valuable and integrate features that an experiment is not able to visualise, there are some probable difficulties with them. Firstly, a small case study is not always helpful in giving techniques and principles for software engineering as the problem in it may differ from the problem in a large case study. Secondly, as there is not enough control over the case study, the results, due to confounding issues, are not always clear.

Kitchenham et al. (1995b) present instructions for arranging and analysing case studies in order to yield significant outcomes. These instructions are firstly, to identify the hypothesis in detail to make clear the measures needed to demonstrate the effect of the methods and secondly, to select the pilot project. A third instruction is to identify the method of the comparison to assess the result of the case study by comparing the results of using the new method against a company baseline and selecting a sister project to contrast with. If the method relates to individual product elements, it could be related randomly to several elements and not to others. Fourthly, it is important to decrease the impact of confounding issues, examples of which are: employing staff who are extremely enthusiastic or unenthusiastic, and using of contrasting application types. Fifthly, the case study needs to be planned and sixthly, it needs to be observed alongside the plan and contrast its development and results with the plan. Finally, the results are required to be investigated and described to summarise what has occurred and to see if the results are significant.

In software engineering, case studies are used in much research. The study of Granja-Alvarez (2004) is based on three real-world projects where a comparative analysis of projects was undertaken and, through this analysis advanced results were able to be achieved in software maintenance. The result derived from this study was that a very high-quality estimate may be gained from *use cases* for software maintenance. Bieman et al. (2001) declare that case studies can illustrate the relationships between design structure and quality attributes such as reliability and maintainability. Their study was carried out on a commercial OO C++ system. They analysed 39 versions of a system to discover if there was a connection between the total number of changes and the design structure in the system. Finally, Briand et al. (1999c) used a commercial case study and investigated the connection between design attributes and the fault-proneness in commercial and student projects. The commercial projects were case studies, whereas the student projects were experiments.

In this research, we used source code archived analysis using multiple versions of several Java OSSs. Our strategy can be declared to be similar to multiple case studies. We used an automated tool, described later, to extract OO metrics from versions of the OSSs. The selection of our approach is justified by the fact that software artefacts can provide a meaningful insight into how professional software developers use and maintain coupling, which in turn provides an insight into the evolution of coupling. Using this approach, we were able to reveal patterns of change in multiple versions of the studied systems.

3.3 Research Objectives

One of the main objectives of this research is to assess how a system changes through the analysis of packages in the system and to compare that data with corresponding results from refactoring the same system. Knowledge of trends and changes within packages is a starting point for an understanding of how effective the original design may have been, how susceptible types of packages may be to change and can also inform our knowledge of facets of software such as coupling and cohesion.

Another objective of the research is to explore the relationship between coupling metrics and the classes removed from multiple versions of several open-source systems, and to empirically explore coupling in these Java systems using coupling metrics, version release times and code warnings. Finally, we aim to explore whether an 80/20 rule exists in Java from coupling metrics over multiple versions of open-source software and, if so, whether that relationship is exacerbated over time.

For each of these objectives, we generate hypotheses that describe and interpret these objectives. Hypotheses can help researchers predict expected results and the direction of their investigation. However, researchers must provide a justification as to why they produce that hypothesis depending on the theoretical aspects. Furthermore, hypothesis testing also requires recognition of suitable data strictly related to the cause and effect of the hypothesis. The data should be divided into two groups, independent and dependent variables. An independent variable refers to a set of data which may have an impact on another set of data (dependent variable) and the dependent variable is a set of data which changes as a result of a change in independent variable. After the independent and dependent variables have been concluded, a fitting statistical test should be detected to precisely test the impact of independent variable on the dependent variable(s).

Hypotheses consist of a *null* hypothesis and an *alternative* hypothesis. A null hypothesis refers to an independent variable which has no significant relationship with the dependent variable(s), and an alternative hypothesis refers to a correlation existing between independent and dependent variables (Field, 2006). Researchers consider that the alternative hypothesis is true, unless the null hypothesis indicates the opposite.

3.4 Data Collection

In this section, a description of the systems under study will be provided along with the definition of the collected metrics.

3.4.1 Systems under Study

The explicit selection criteria for systems was that first, they all had to be entirely Java; second, sufficient versions were available (for a longitudinal study) and third, they should consist of a mix of application types. Systems were selected in terms of ‘number of downloads’ ordered from sourceforge.net. The selection process thus resulted in many systems being rejected from candidate systems identified (because they were either a mix of different languages and/or did not contain multiple versions for download).

1) **Velocity:** A template engine allows web designers to access methods defined in Java. Velocity began with 224 classes and 44 interfaces. In the latest version, it had 300 classes and 80 interfaces.
2) **Jasmin:** A Java assembler takes ASCII descriptions of Java classes and converts them into binary Java .class files suitable for loading into a Java Virtual Machine. The system is comprised of 5 versions. It started with 5 packages and 110 classes in the first version and had 5 packages and 130 classes by the latest version.
3) **pBeans:** Provides automatic object/relational mapping (ORM) of Java objects to database tables. The system comprised of 10 versions, with 4

packages and 36 classes in the first version with 10 packages and 69 classes in the latest version.

4) **SmallSQL:** A Java DBMS for Java desktop applications. It has a JDBC 3.0 interface and offers many ANSI SQL 92 and ANSI SQL 99 features. The system comprised of 8 versions. It started with 130 classes in the first version and had 177 classes in the latest version.

5) **JWNL:** A Java API for accessing the WordNet relational dictionary. WordNet is widely used for developing NLP applications and allows developers to use Java for building NLP applications. The system comprised of 5 versions. It started with 11 packages and 95 classes in the first version with 15 packages and120 classes in the latest version.

6) **DjVu:** Provides an applet and desktop viewer Java virtual machine. The system is comprised of 8 versions. It started with 12 packages and 77 classes in the first version with 14 packages and 79 classes in the latest version.

7) **Asterisk:** A Java system consists of a set of Java classes that allow you to easily build Java applications that interact with an Asterisk PBX Server. It supports the FastAGI protocol and the Manager API. This system includes 6 versions. It started with 12 packages and 222 classes in the first version and ended with 14 packages and 277 classes in the final version.

Table 3.1 shows the data for the seven systems under study.

Table 3.1 Systems under study

System	Number of versions	Number of Packages	Number of classes
Velocity	9	28-39	224-300
Jasmin	5	5	110-130
pBeans	10	4-10	36-69
SmallSQL	9	1-3	130-177
JWNL	5	11-15	95-120
DjVu	8	12-14	77-79
Asterisk	6	12-14	222-277

3.4.2 Software Metrics Definition

OO metrics usually capture properties of OO systems such as cohesion, inheritance, encapsulation, polymorphism, size or coupling (Fenton and Pfleeger, 2002). It is important for a researcher to analyze whether the software metric used is well defined and valid (Fenton and Pfleeger, 2002). This guarantees that the software metric(s) truly measure(s) the attribute(s) of a product, process or project which it states to measure. For this Thesis, we adopted an automatic approach for data collection using the JHawk tool (JHawk, 2008). JHawk was used to extract OO metrics from versions of the systems under study. It uses static analysis of source code to extract a variety of OO metrics stated in the literature. We justify our selection of the tool on the basis that it was used by other researchers in the field of SE (Arisholm et al., 2004). The following is a description of the metric definitions used throughout this Thesis:

1) Response for a Class (RFC): This metric is the same as that defined by Chidamber and Kemerer (1994) and measures the response set of a class. The RFC is defined as the set of methods that can be potentially executed in response to a message received by an object of that class.
2) Message Passing Coupling (MPC): The number of messages passed among objects of a class.
3) PACK: Number of imported packages.
4) Number of EXTernal methods called (EXT): The more external methods that a class calls the more tightly bound that class is to other classes.
5) Fan In (FIN): FIN of a function is the number of unique functions that call the function.
6) Fan Out (FOUT): FOUT counts the number of distinct non-inheritance related class hierarchies on which a class depends.

We also collected for each class the total number of methods (private, protected and public) and the lines of code (LOC) in each class as size measures. We also collected one time based metric which is the time interval between each version release.

3.5 Data Analysis

Measurement is an essential concept in engineering. The conclusions of any empirical study are built on the values measured on research variables. Therefore, it is fundamental to consider the quality of the measurement and consequently their conclusions. Statistics is a tool that can assist researchers in giving the quantitative estimate of the probable truth of the conclusions.

In this Thesis, we used three correlation coefficient analyses (Pearson’s, Kendall’s and Spearman’s) to investigate the relationship between our variables. Researchers are usually interested in measuring the relationship between two or more variables. Field (2005) identifies correlation as a measure of the linear relationship between variables. These variables may relate to each other in one of the following ways: they may be positively related, they may be not related at all, or they may be negatively related. Field introduces two major calculations for correlations: Pearson’s Correlation and Spearman’s Correlation. Pearson’s Correlation is a parametric statistical test that needs interval or ratio data and is normally distributed; Spearman’s Correlation is a non-parametric test for ranked data so it can be applied to data that is not normally distributed.

3.6 Summary

In this chapter, we presented a discussion of the methods used to conduct our empirical research including, the design of the study, a description of the sample systems selected, the definition of software metrics used, data collected and statistical techniques used.

The following chapter presents an empirical study that investigated longitudinal trends in changes to an OSS. We consider the trends in versions of the OSS, with respect to regular maintenance changes. These changes include the added classes, methods, attributes and lines of code. The relationship between these changes and refactoring data is considered as well.

CHAPTER 4. PACKAGE EVOLVABILITY AND ITS RELATIONSHIP WITH REFACTORING

4.1 Introduction

Central to the aim of the Thesis is to uncover traits in OSS from an evolutionary perspective. Project managers and developers alike have a keen interest in minimising the amount of code 'decay' that usually occurs as a system ages. It is also important that different levels of evolutionary abstraction are considered to give different perspectives on the same systems; equally, that different types of change (corrective, perfective or adaptive) are explored.

In this Chapter, we therefore consider trends in versions of the 'Velocity' OSS, with respect to added classes, methods, attributes and lines of code and the relevant enclosing packages. To support our analysis of change type, we also look at empirical refactoring data for the same system and associated trends for two other Java OSSs, namely PDFBox and Antlr.

It is suggested that if the set of *regular* (i.e. essential) maintenance changes reveal specific characteristics, then a set of specific refactorings will also reveal similar features. Results showed an interesting inconsistency between trends in those regular changes made to the system studied and those as part of a specific set of changes according to refactorings specified in Fowler (1999).

The remainder of the chapter is organised as follows. In the next section, the motivation for the undertaken study is presented and in Section 4.3, details of the data collected is provided addressing three research questions. In Section 4.4, the research questions are evaluated through analysis of the data over the nine versions of the system. Section 4.5 provides a discussion of the refactoring relationships from two points of view: the relationship between the refactorings and the changes in the new added classes, LOC, methods and attributes, and the relationships among the considered refactorings. The results are discussed in Section 4.6, and finally a summary for the study and its results given in Section

4.7. We note that the research described in this chapter was first published by Mubarak et al. (2007).

4.2 Motivation and Related Issues

A software system is modified and developed many times throughout its lifetime to maintain its effectiveness. In general, it grows and changes to support increases in information technology requirements. From a research perspective, we know a reasonable amount about facets of OO and procedural system evolution (Belady and Lehman, 1976; Bieman et al., 2003; Arisholm and Briand, 2006). However, it is less well-understood whether changes at the package level exhibit any specific trends. The benefit of a study that explores changes at this level is clear. Understanding changes at higher levels of abstraction may give a project manager a much more general idea of likely future maintenance or refactoring opportunities. In particular, such a study may also be able to focus developer effort in specific areas of packages susceptible to large numbers of changes. An additional topic of concern to OO practitioners and researchers is the relationship between maintenance as part of the system's development and that related to refactoring (Fowler, 1999; Mens and Tourwe, 2004).

From an empirical point of view, the relationship between OO classes and packages is not well defined. Ducasse et al. (2005) suggest that it is necessary, for the re-engineering and development of OO systems, to recognise and investigate *both* sets of classes and packages. Ducasse et al. (2004) suggest that the cost of modifying a program may be influenced by the relationship between packages and their enclosed classes. In terms of the architecture of a system, Bieman et al. (2001) found that classes belonging to a design pattern were the most change-prone classes in a system (this might also suggest that change-prone classes are implemented by design patterns). Finally, Demeyer et al. (2000) identified refactoring indicators when comparing different releases of a software system. They used four heuristics to find refactorings; each was identified as a mixture of change metrics. In this study, we will investigate the changes in the packages level in an OSS by considering the changes in the added classes, number of the line of codes, the methods and the attributes in the packages basis. We also consider relationships between trends in changes at the class level with refactoring data extracted using a bespoke tool.

4.3 Empirical Investigation

The main objective of the research described is to assess how a system changes through the analysis of packages in the system and to compare that data with corresponding results from refactoring the same system. Knowledge of trends and changes within packages is a starting point for an understanding of how effective the original design may have been and how susceptible types of packages may be to change. It can also inform our knowledge of facets of software such as coupling and cohesion.

4.3.1 The System under Study

To achieve our objectives, a case study approach was adopted using multiple versions of an evolving system. This system was a large OSS called ‘Velocity’ – a template engine allowing web designers to access methods defined in Java. Velocity began with 224 classes and 44 interfaces. In the latest version, it had 300 classes and 80 interfaces.

The data analysed was the change data on a package basis for nine versions of the system. The study investigated patterns in change over those nine versions through three research questions. In other words, certain features were investigated about how a system evolved based on what we believed should happen to a system over time. The research questions were supported by statistical analysis.

Table 4.1 shows the changes in the number of packages and new classes added to the system over the course of the nine versions.

The data for each package is categorised in several columns, and each column contains the changes that have occurred to the packages since the previous version. These columns are structured as follows for each package:

1. Number of classes where lines of code decreased, number of attributes and number of methods decreased.
2. Number of classes where lines of code decreased, number of attributes decreased and number of methods stayed the same.
3. Number of classes where lines of code decreased, number of attributes stayed the same and number of methods decreased.
4. Number of classes where lines of code decreased, number of attributes stayed the same and number of methods stayed the same.
5. Maximum decrease in the lines of code for that transition.
6. Maximum decrease in number of attributes for that transition.
7. Maximum decrease in number of methods for that transition.
8. Number of new classes added during that transition.

Table 4.1 The number of packages and new classes over the course of 9 versions

Version	Number of packages	Number of new classes
1st	28	788
2nd	32	1116
3rd	38	17
4th	42	11
5th	36	2032
6th	39	45
7th	39	297
8th	38	1274
9th	39	1386

We also collected data for the corresponding increases, obtained by replacing the word ‘decreased’ with ‘increased’ in the above list of eight columns. We looked into the changes over the course of these nine versions by investigating the changes occurring in the individual packages. There are many ways to measure the changes in the packages; however, we determined these changes by assessing the changes in the number of added lines of code, the number of added methods, and the number of added attributes. Therefore, for each version, we collected the number of added classes, lines of code (LOC), methods and attributes. (Henceforward, we define a LOC as a single executable statement; we therefore disregard comment lines and white space from calculation of LOC.)

4.3.2 The Research Questions

The trends of changes in the packages for the OSS are inspected through three research questions. These questions investigate the trends of the added classes, increases in the LOC and the increases in the number of attributes and methods in the packages across the nine versions of the system. The research questions are as follow.

- **RQ1**: Does the number of new classes over the course of nine selected versions increase constantly? This question is based on the notion that a system will grow over time in a constant fashion in response to regular changes in requirements.

- **RQ2**: Is the increase in LOC over the course of the nine versions constant? This question is based on the assumption that the change in LOC over the nine versions will always increase due to evolutionary forces.

- **RQ3**: Is the increase in the number of attributes and methods in a package constant across the versions of a system? This question is based on the assumption that the change in the number of attributes and methods will increase consistently over time in response to constant changes in requirements.

4.4 Data Analysis

We determined the changes in the packages by assessing the changes in the number of added lines of code, the number of added methods, and the number of added attributes. In order to assess our research questions, we organised our collected data in a table and a figure for each question.

4.4.1 Research Question 1 (RQ1)

The first research question investigates whether the numbers of new classes over the course of nine selected versions increase constantly. Table 4.2 shows the number of packages in each of the nine versions, the number of new classes across those packages, the number of new classes in total, the maximum increase in classes and the package name where that increase took place. In each of the nine versions, new classes were added to packages and the number added varied significantly from one version to another. Between versions three and four and six and seven, relatively little change can be seen, while the peak of added classes is reached in the fifth version with 2032 new classes added. Clearly, the addition of classes to this system over the versions investigated was not constant. Interestingly, the version with the highest number of new classes was also accompanied by a drop in the number of packages (from 42 to 36). Equally, some of the largest additions to classes were made after only minor changes to the numbers of packages. Both effects may possibly be due to classes being moved around in the same package and simply renamed.

A feature not immediately apparent from the data in Table 4.2 is the peak and trough effect of this data. A graph was therefore used to present the changes in the number of new added classes (Figure 4.1). We suggest that this trend is symptomatic of a burst of developer change activity followed by a period of relative stability and accumulation of new requirements, before another burst of change activity. A closer view of the data shows us that this increase is not always in the same packages for each version, and the packages themselves do not have the same number of classes, so this may clarify the differences in the numbers of added classes across the nine versions. Furthermore, these differences may be affected by external reasons associated with: the developers' experiences, the product users and their requirements and the period of time separating each of the versions. For RQ1, we conclude that the number of new classes over the course of the nine versions increases at an *inconsistent* rate, rather than remaining constant. It is not the case that there is constant addition of classes to the Velocity system over the nine versions investigated; RQ1 cannot thus be supported.

Table 4.2 Packages and the new classes over the course of 9 versions

Version	No. of packages	No. of new Classes	Max inc. in new classes	Package name
1st	28	788	176	Editor
2nd	32	1116	207	Java
3rd	38	17	5	Core
4th	42	11	3	Javadoc
5th	36	2032	329	Debuggerjpda
6th	39	45	13	Openide
7th	39	297	92	Core
8th	38	1274	357	Web
9th	39	1386	217	Core

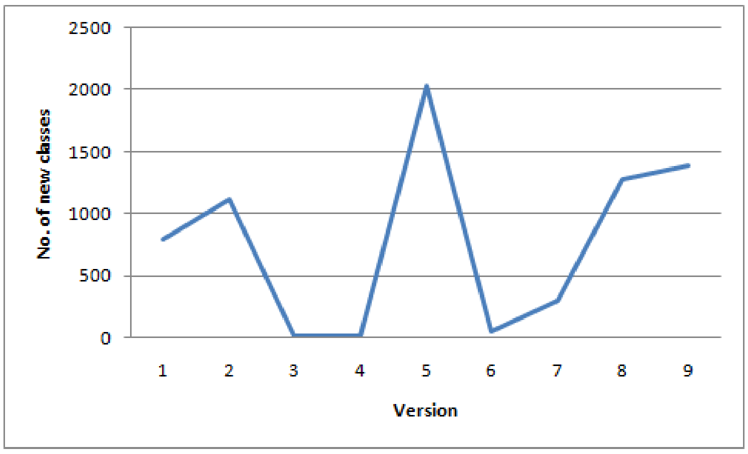

Figure 4.1 Line chart of new classes added to the packages over the 9 versions

4.4.2 Research Question 2 (RQ2)

The second research question is whether the increase in LOC over the course of the nine versions is constant. To investigate RQ2, the ‘maximum’ increase in the number of LOC among all the versions was used. The data is presented in Table 4.3. It can be seen that there are increases in LOC over the course of the versions, but these increases fluctuate wildly. Interestingly, the Core and Vcscore packages were the packages that saw the maximum increases in LOC for five of the versions. The Core package is the only common package in Table 4.2 and Table 4.3, suggesting that the addition of a large number of classes does not necessarily imply the addition of a correspondingly large number of LOC. One explanation for this feature might simply be that one class has been split into two (c.f. the ‘Extract Class’ refactoring of Fowler (1999)).

Scrutiny of the data indicates that the increase in the number of LOC in a package is not always the same for each version; it varies from one version to another across the nine versions. From the data under study it can be seen that in addition to this increase in the number of LOC there is always a decrease in the same number for each package. In other words, there are always some added LOC and at the same time removed LOC also. Furthermore, the collected data presents the maximum increase in the number of LOC among all the transitions of the classes, this maximum may even be an outlier. All of these previous reasons may explain the differences in the change of the number of LOC in addition to the external reasons explained in the prior research question.

Figure 4.2 confirms that the increases in the number of LOC over the course of the nine versions fluctuate across versions. Again, the peak and trough effect is apparent from the figure. The most significant changes to Vcscore appear in the first five versions and those of Core appear in the seventh and eighth versions; RQ2 cannot be supported either.

Table 4.3 Max. increase in the number of LOC over the course of the 9 versions

Version	Max inc in LOC	Max inc in LOC among all the packages	The name of the package
1st	3955	547	Core
2nd	5077	686	Form
3rd	889	226	Vcscore
4th	910	320	Javacvs
5th	6985	995	Vcscore
6th	1109	111	Vcsgeneric
7th	369	71	Core
8th	6418	1854	Core
9th	6743	1236	Schema2beans

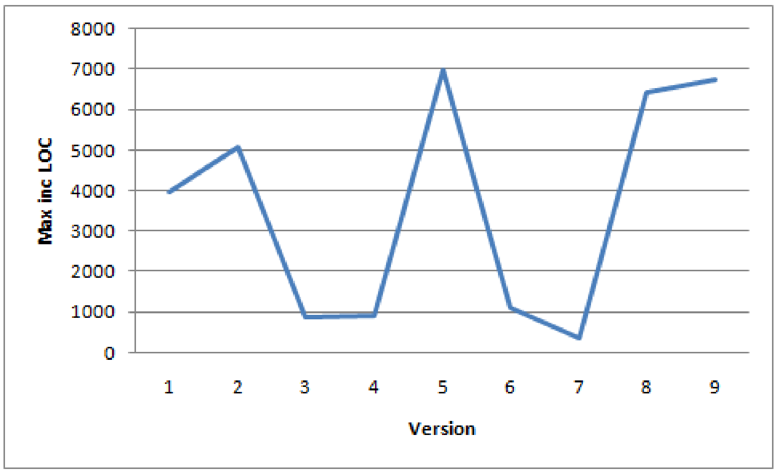

Figure 4.2 Line chart of the max increase in the number of LOC (9 versions)

4.4.3 Research Question 3 (RQ3)

The third research question is whether the increase in the number of attributes and methods in a package is constant across the versions of a system. For this research question, the maximum increase in the number of attributes, and the maximum increase in the number of methods for each version were used. This data is presented in Table 4.4 which shows that over the course of the nine versions there are consistent increases in the number of attributes (A) and number of methods (M). However, these increases vary from one version to another. The largest increase in the number of the attributes and methods is at version five. These differences in the changes may occur because of the differences in the number of classes in each package or because of the different structure for each package. In addition, they may be affected by the reasons suggested in the last research question related to the increases and decreases in the same variable, considering the maximum number for variable in the collected data, as well external factors.

Once again, two packages dominate Table 4.4 - those being Core and Vcscore (seven of the eighteen entries in columns 4 and 7 relate to these two packages). As for Table 4.3, the maximum increase in methods occurs at earlier versions for Vcscore and towards later versions for Core.

Table 4.4 Summary of the increase in attributes and methods over the 9 versions

Version	Inc in A	Max Inc in A	Package name	Inc in M	Max Inc in M	Package name
1st	153	26	Core	228	36	Vcscvs
2nd	262	49	Form	335	84	Vcscore
3rd	25	7	Jndi	46	11	Vcscore
4th	24	7	Diff	22	6	Diff
5th	325	51	Form	489	70	Vcscore
6th	39	10	Debuggercore	73	14	Openide
7th	17	4	I18n	29	6	Core
8th	238	57	Core	371	150	Core
9th	226	34	Java	378	76	Xml

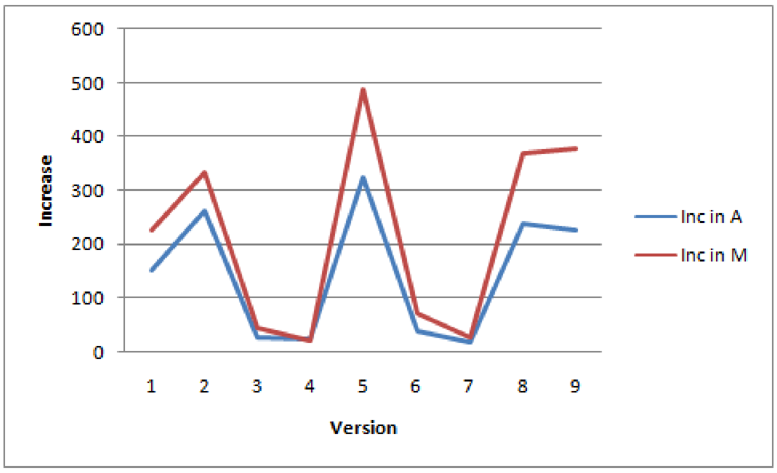

Figure 4.3 Inc. in attributes and methods

Figure 4.3 shows that the number of attributes and number of methods both increase during the course of the nine versions, but at a fluctuating rate. Version four shows that more attributes were added than methods; the pattern for all other versions is the opposite. In contrast to the previous analysis, Figure 4.4 shows that version eight appears to be the source of the largest increase in methods. In keeping with the results from RQ1 and RQ2, we conclude for RQ3 that the increase in attributes and methods is *not* constant across the nine versions investigated. While the results so far give a fairly intuitive understanding of how a system might evolve, what is not so clear is the relationship between the ‘regular’ set of changes as we have described them, and the opportunities for undertaking a set of changes such as those associated with refactoring techniques (Opdyke, 1992; Tourwe and Mens, 2003). These are both interesting and potentially fruitful areas of refactoring research as well as challenges facing the refactoring community (Mens and van Deursen, 2003).

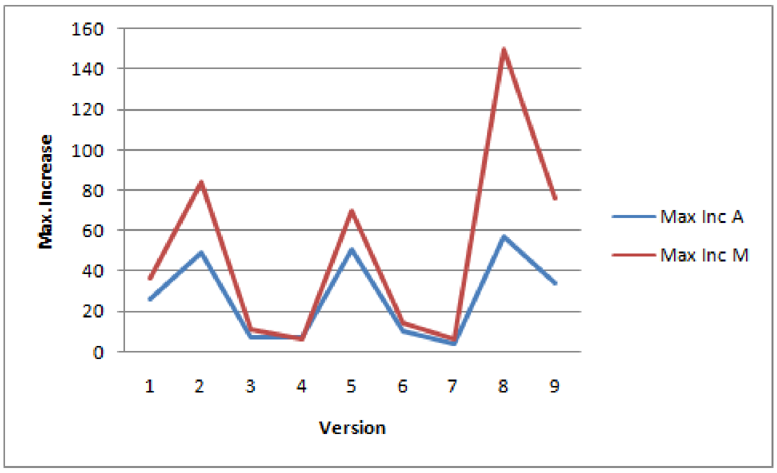

Figure 4.4 Max. inc. in attributes and methods

4.5 Refactoring Relationships

Beck (1999) suggests that a developer should refactor ‘mercilessly’ and hence consistently. We would therefore expect refactorings for the Velocity system to be consistently applied across all versions. In this section, we first investigate the refactoring in Velocity system and then in other two OSSs - PDFBox and Antlr.

4.5.1 Velocity

For Velocity system, we analysed fifteen refactorings across the nine versions. These refactoring are presented in the first column in Table 4.5. They were collected by a software tool as part of a full study of refactoring in seven Java OSS systems by Advani et al. (2006). The fifteen refactorings were chosen by two developers with industrial experience and reflected, in their opinion, in consultation with Fowler’s text (Fowler, 1999), the common refactorings likely to be made by developers over the course of system’s life. As such, refactorings embracing inheritance, encapsulation, movement of class features and their addition and removal, all are included amongst the fifteen refactorings.

The data presented in Table 4.5 is the number of the refactoring in each of the nine versions for the Velocity system. It can be seen that versions 3, 5 and 6 are the main points when refactoring effort was applied to the Velocity system (these columns are bolded). In versions 1, 4, 7 and 8, zero refactorings were applied to this system.

Figure 4.5 shows Table 4.5 in graphical form (with the ‘per version sum’ of the fifteen refactorings on the *y*-axis). The figure shows that refactoring effort is applied most significantly at one version (in this case version 3) and thereafter a peak and trough effect can be seen. Comparing the trend in Figure 4.5 with that in Figures 4.1-4.4 suggests that the majority of the refactoring effort occurred *between* versions where significant changes in classes, LOC, methods and attributes took place. Version 3, with the most refactorings effort across the nine versions, is a trough in terms of these added features. Conversely, version 5 from Table 4.5 shows significant refactoring effort to have been applied, coinciding with large changes in the aforementioned features. Version 6 activity (again a trough in terms of Figures 4.1-4.4) also shows relatively large amounts of refactoring effort.

Table 4.5 Refactorings for the Velocity system across 9 versions

No.	Refactoring	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
1.	AddParameter	0	0	**14**	0	**1**	**2**	0	0	1
2.	EncapsulateDowncast	0	0	**0**	0	**0**	**0**	0	0	0
3.	HideMethod	0	2	**1**	0	**0**	**1**	0	0	0
4.	PullUpField	0	0	**4**	0	**2**	**4**	0	0	0
5.	PullUpMethod	0	4	**13**	0	**24**	**5**	0	0	9
6.	PushDownField	0	0	**0**	0	**7**	**0**	0	0	0
7.	PushDownMethod	0	0	**1**	0	**1**	**0**	0	0	4
8.	RemoveParameter	0	0	**3**	0	**1**	**0**	0	0	3
9.	RenameField	0	3	**14**	0	**1**	**2**	0	0	3
10.	RenameMethod	0	5	**11**	0	**15**	**14**	0	0	10
11.	EncapsulateField	0	5	**4**	0	**0**	**0**	0	0	0
12.	MoveField	0	0	**18**	0	**1**	**2**	0	0	0
13.	MoveMethod	0	3	**16**	0	**3**	**3**	0	0	2
14.	ExtractSuperClass	0	1	**3**	0	**8**	**1**	0	0	2
15.	ExtractSubClass	0	0	**0**	0	**1**	**0**	0	0	0

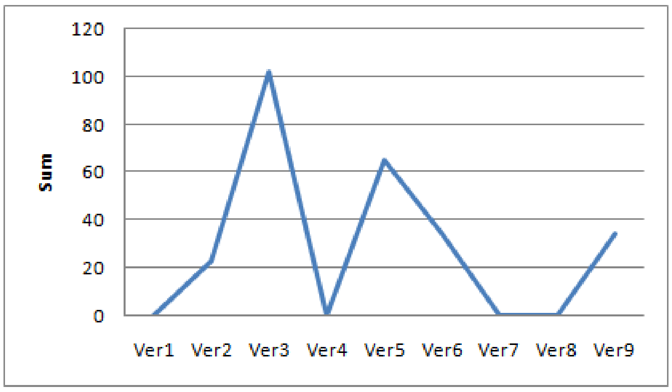

Figure 4.5 Refactorings in the 9 versions of Velocity

A number of conclusions can be drawn from this analysis. Firstly, it is clear that developers do not seem to refactor consistently across the versions of the system studied (Velocity) as there is not any refactoring for some versions. Secondly, there is some evidence of peaks in refactoring effort happening simultaneously with large changes in classes, LOC, methods and attributes, while refactoring seems to take place mostly in a version *after* a peak of the same type of changes. (One of the claims by Fowler (1999) as to why developers do not do refactoring is that they simply do not have the time.) Finally, in the previous analysis, and from Figure 4.5, it can be noticed that the majority of regular change activity applied to the system is not applied during the initial versions. We considered the time interval between two versions as a variable in our study in Chapter 6.

The first question that naturally arises is why refactoring changes tend to follow the regular changes applied to a system? After all, it is quite feasible for refactoring to be carried out at the same time as other changes (there is limited evidence of this occurring from the data). Moreover, the opportunity for refactoring often arises as part of other maintenance activity and we would thus expect developers to spot opportunities for refactoring as they undertake other work on a system. There is one relatively straightforward explanation for this phenomenon. All of the fifteen refactorings in Table 4.5 are semantics-preserving and do not explicitly add large numbers of classes, LOC, methods or attributes as part of their mechanics. For example, the ‘Move Field’ and ‘Move Method’ refactorings would have no net effect on the number of fields or methods in a system, on a package basis. Simple renaming refactorings such as ‘Rename Field’ and ‘Rename Method’ do not add any LOC to the system either. Equally, none of the inheritance-related refactorings explicitly add LOC to a system.

One further suggestion as to why refactoring occurs at different versions is that after a burst of regular maintenance effort and a new version being released, the decay to the system that those changes have caused may need to be remedied. In other words, after a concerted effort to modify the system through regular maintenance, developers may feel that only *then* is refactoring necessary. However, this does not explain why for the Velocity system there is significant refactoring effort in version 5 occurring together with a large set of changes in terms of added classes, LOC, methods and attributes. One explanation could be that developers refactor during the course of normal maintenance but without explicitly recognising it as refactoring. In other words, they may tidy up the code after completing the changes in the system classes, LOC, methods or attributes. We could hypothesise that while for Velocity (and the refactorings we have extracted) refactoring effort is not applied consistently, there are two key occasions when, consciously or sub-consciously, it *is* applied.

One aspect of the analysis that we have not yet considered is the relationship between the refactorings from Table 4.5.

Figure 4.6 shows the sum of refactorings across all nine versions of the Velocity system (the numerical data for this graph is exactly that in Table 4.5). Each line in the graph represents the sum of each refactoring for a single version. So, for example, refactoring five (Add Parameter) when taken in totality is a common refactoring across most versions (at least five); the graphs at refactoring 5 show simultaneous peaks. Equally, refactoring ten (Rename Method) can be considered as a popular refactoring in each of the versions. For the fifteen refactorings, a clear trend of peaks and troughs in the fifteen refactorings can be seen. In other words, there is a trend in the propensity of refactorings to occur in ‘parallel’ (at the same time). Figure 4.6 thus illustrates the strong bond between the fifteen refactorings.

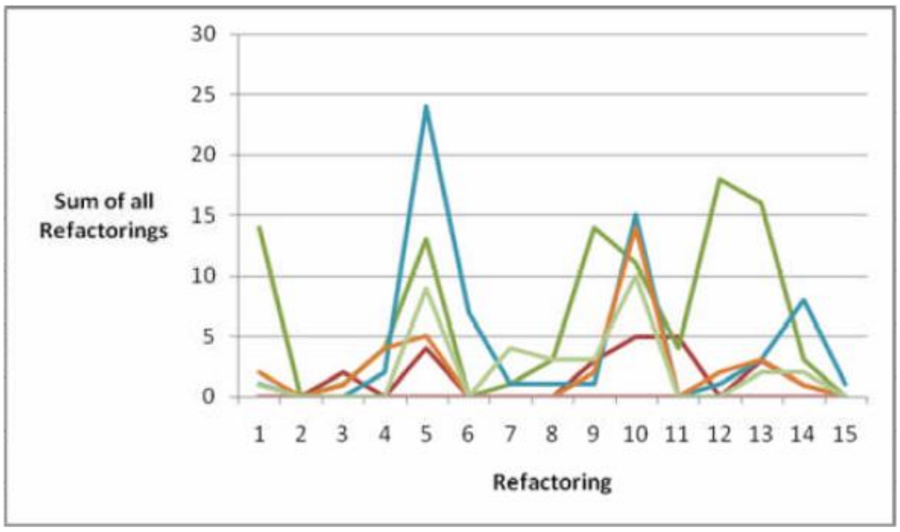

Figure 4.6 'Peak and trough' effect of refactorings for Velocity

Two notable exceptions to the trend of refactorings follow peaks and troughs apply to refactoring one (Add Parameter) and twelve (Move Field). At times, there are large numbers of this refactoring in a particular version and very few other refactorings in the same corresponding versions. A simple explanation may account for this trend. They are both refactorings that are used by the mechanics of many other refactorings. They are also two refactorings that a developer may undertake in the course of regular maintenance for example, to fix a fault without the use of any other refactorings. In other words, they can both act as stand-alone refactorings in contexts other than that of refactoring.

4.5.2 PDFBox and Antlr

The question we could then ask is whether refactoring effort is consistent in terms of the versions where it is undertaken, and whether a similar trend in refactoring appears in other systems. In order to investigate that, we analysed the refactoring data from other two systems; PDFBox and Antlr.

Figure 4.7 shows the versions where refactorings were undertaken for the PDFBox system. Versions 3 and 6 appear to be where the majority of the refactoring effort was invested. Although we do not have the dataset of regular maintenance changes applied to the PDFBox system, it is interesting that a peak and trough effect is clearly visible for this system as well as for Velocity.

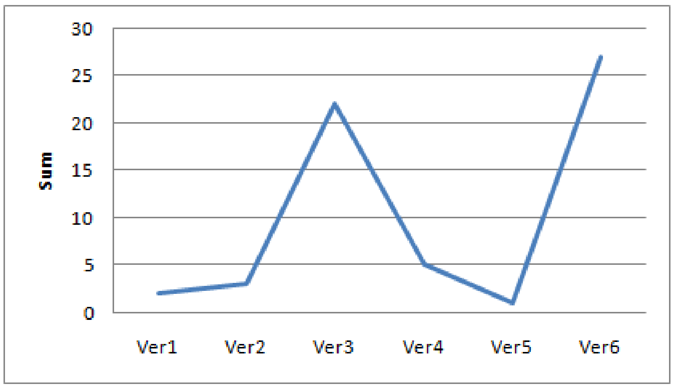

Figure 4.7 Refactorings for PDFBox

Figure 4.8 shows refactoring trends for the Antlr system. Version two appears to be the one which most refactoring effort was invested in, supporting the view that relatively more refactoring seems to be undertaken at early versions of system's life (but not at its inception). It is interesting that across all three systems, version one seems to have been the subject of virtually no refactoring effort. One explanation might be that version one is simply too early in the life of a system for refactoring effort to be applied. On the other hand, it appears that version two or three is when the majority of refactoring occurs. The question that then arises is whether the numbers of each type of refactorings in each of the three systems were similar?

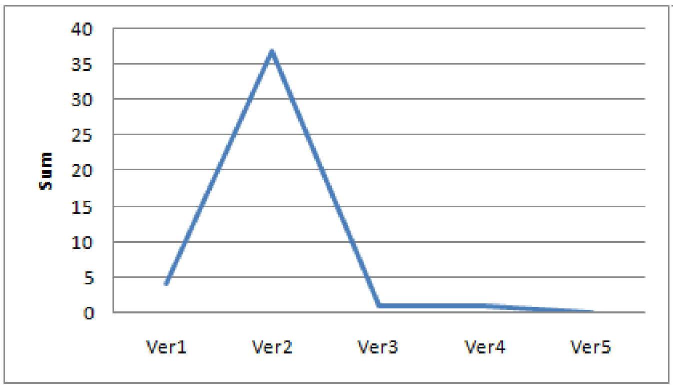

Figure 4.8 Refactorings for Antlr

Inspection of the raw data reveals a common trend for refactoring 1 (Add Parameter) and refactorings 9, 10, 12 and 13 (Rename Field, Rename Method, Move Field and Move Method). We hypothesise that these types of refactoring have been applied relatively more frequently than any of the other fifteen because they 'tidy up' a system with relatively little effort being required. After a significant amount of maintenance effort has been applied to a system, minor modifications are bound to be necessary. This may further explain why there is no coincidence between regular maintenance effort and that of refactoring. In the analysis of changes made at the package level, a significant number of methods and attributes were added over the versions studied.

4.6 Discussion

Based on the refactoring evidence, we could claim that the five stated refactorings were a direct response to the problems associated with the addition of so many attributes and methods. For example, the motivation for the 'Move Field' refactoring is when '*a field is, or will be, used by another class more than the class on which it is defined*'. In such a case, the field needs to be moved to the place '*where it is being used most*'. Equally, the 'Move Method' refactoring is applicable when: '*A method is, or will be, using or used by more features of another class than the class on which it is defined*'. For the Velocity system, the large number of these two refactorings at version three suggests that the correspondingly large number of fields and methods added were the cause of required subsequent refactoring. In other words, simple refactorings may have been undertaken to remedy the problems associated with such an intense set of added fields and methods.

We also note that these two refactorings were popular *across all three* systems studied (and at specific points), which adds weight in support of this argument. The same principle applies to simple renaming of fields and methods. It is perfectly reasonable to suggest that when large numbers of attributes and methods have been added to a system, a certain amount of refactoring may be necessary subsequently to disambiguate and clarify the role and meaning of those fields and methods. Fowler (1999) suggests that the 'Move Method' refactoring is the '*bread and butter of refactoring*'. Equally, 'Move Field' is the '*very essence of refactoring*'.

Similarly, Fowler (1999) reveals an interesting point about the 'Rename Method' refactoring: '*Life being what it is, you won't get your names right the first time*'. One explanation for the lack of the more 'structurally-based' refactorings (i.e. those that manipulate the inheritance hierarchy) in the systems studied might be that the package access provides the necessary inter-class access that inheritance might otherwise provide. The 'Extract Subclass' and 'Extract Superclass' refactorings would fall into this category. One final point relates to why versions two and three were the source of the most refactoring effort (as opposed to later versions of the system across all three systems). One explanation is that when a system is at early stages of its lifetime, the design documentation is more likely to be up-to-date. Consequently, the system is relatively easy to modify from a refactoring perspective. As the system ages, increasing amounts of effort and time needs to be devoted to changes as the code 'decays'.

There are a number of threats to the validity of the study that need to be considered. One threat is that we have only considered a relatively small sample of systems to investigate. In defence of this threat, we accept that a larger sample of systems might demonstrate that the results in this chapter are more generalisable to the population of systems (external validity). However, the same criticism could be made of a study with double the number of systems studied, for example. A further threat to the validity of the study is that we have only considered fifteen refactorings from the 72 stated in Fowler (1999). We have also only considered a relatively small number of versions of each system; again, in defence of this claim, we chose the most number of versions available at the time the research was being undertaken. One final threat to the validity of the study relates to the time gap between each version of a system. We have assumed, so far, that there is an equivalent time gap between versions and hence that, other things remaining equal, there is a reasonable chance of the same number of refactorings being undertaken between each version.

Table 4.6 shows the time gap in months (m) and days (d) between the nine versions of the Velocity system and the total number of refactorings that were identified in that time - the totals are calculated by summing the individual columns of Table 4.5. (For the sake of argument, we assume a month to be 30 days duration.) Table 4.6 shows that there is a wide variation in times between versions of the Velocity system. The minimum gap is 8 days and the maximum gap 8 months, 8 days. What is most interesting and noteworthy from Table 4.6 is that there is no clear pattern or proportionality with the number of refactorings based purely on the version time gaps. In other words, the length in time between versions seems to have no bearing on the number of refactorings extracted by the tool and undertaken by the developers of this system. For example, the 8 month, 8 day gap between version 7 and version 8 realised zero refactorings. Equally, the 10 days between version 2 and 3 realised the highest number of (102) refactorings. Inspection of the Velocity change logs detailing the changes between versions revealed a mixture of patches, bug fixes and new requirements. It would therefore seem that refactoring may be motivated by factors other than time *per se*. The amount of developer effort invested into the system between versions, for example, may be a more significant factor than time. A finer-grained analysis of exactly at what date and time the refactorings were undertaken (i.e. a timestamp approach) as well as some indication of effort on the part of the developers might also provide a greater insight and reveal more informative patterns in the refactorings; we leave this detailed aspect of the analysis for future work.

Table 4.6 Duration between each version and associated refactorings (Velocity)

Version	Version gap	Refactorings
Ver1	-	0
Ver2	8d	23
Ver3	10d	102
Ver4	5m 28d	0
Ver5	1m 21d	65
Ver6	6m 28d	34
Ver7	15d	0
Ver8	8m 8d	0
Ver9	7m 9d	34

4.7 Summary

The goal of the research in this chapter was to investigate how a system evolved at the package level and this goal was achieved through the use of a case study. A set of three research questions investigated trends in changes of nine versions of a Java OSS. A bespoke tool was written to extract data relating to changes across those nine versions. An interesting 'peak and trough' effect trend was found to exist in the system studied at specific versions of the system, suggesting that developer activity comprises a set of high and low periods. A contrast was found between those regular changes and those associated with refactoring activity.

The results address a hitherto unknown area - that of the relationship between regular changes made to a system as part of maintenance and that of refactoring. While the study describes only a limited sample of systems and evidence of the peak and trough effect is similarly restricted (both threats to study validity), we view the research as a starting point for further replicated studies and for an in-depth and generalised analysis of coupling/refactoring, both inter- and intra-package.

Since the focus of this Thesis is on trends in coupling at the package level longitudinally, the next chapter will explore whether the extent of coupling influenced the removal of classes from a system. Moreover, we investigate whether size was an influence on removed classes, and whether these removed classes were changed significantly before being removed.

CHAPTER 5. AN EMPIRICAL STUDY OF "REMOVED" CLASSES

5.1 Introduction

In the previous chapter, an investigation of trends in changes to an OSS was conducted. These trends were considered with respect to added classes, LOC, methods and attributes. In addition to this set of maintenance changes, the applied refactorings were investigated in terms of their relationships with those changes.

One aspect of evolution detailed in Chapter 4 and a key observation was therefore the dynamic nature of systems and, in particular, the tendency for removal of classes as a system evolved.

Removal of classes can occur for range of reasons. One plausible reason might be that a class is excessively coupled and therefore needs to be amalgamated and dispersed within the classes to which it is coupled. Equally, a class might be doing very little 'work' and as such can easily be removed from the system with minimal disruption to the rest of the system. In this chapter, an empirical study of coupling and data related to classes removed from multiple versions of four systems are described.

Coupling is a necessary feature of OO systems; ideally, classes with excessive coupling should be either refactored and/or removed from the system. However, a problem that immediately arises is the practical difficulty of carrying out the removal of such classes due to the many coupling dependencies they have; it is often easier to leave classes where they are and work around the problem. In this chapter, we answer three related research questions. First, are classes removed from the system lowly or highly coupled relative to other classes in the same package? Second, are the same classes excessively large compared with the remaining classes in the package? Third, are removed classes changed frequently before they are removed? Results showed a strong tendency for classes with low fan-in (incoming coupling) and fan-out (outgoing coupling) to be candidates for removal. Evidence was also found of class types with high imported package and external call functionality being removed. Finally, size, in terms of methods and lines of code did not seem to be a contributing factor to class removal. The research addresses an area that is often overlooked in the study of evolving systems, notably the characteristics and features of classes that disappear from a system.

The chapter is organised as follows: Section 5.2 describes the motivation for the research and related previous work. In Section 5.3, the systems under study are introduced together with an overview of the metrics collected. Section 5.4 presents an analysis of the data collected; Section 5.5 provides a discussion of the points raised by the study and finally, a summary and future research are presented (Section 5.6). We note that the research in this chapter was first published by Mubarak et al. (2008a).

5.2 Motivation and Related Issues

Excessive class coupling has often been related to the tendency for faults in software (Briand et al., 1997). It is widely believed in the OO software engineering community that excessive coupling between classes creates a level of complexity that can complicate subsequent maintenance and potentially lead to the seeding of further faults. In practice, a class that is highly coupled with many other classes is an ideal candidate for re-engineering or removal from the system to mitigate current and potential future problems. Moreover, a highly coupled class is, other things remaining equal, likely to have grown to be a relatively large class, making it even more suitable theoretically for removal from the system. The paradox that immediately arises, however, is that it is often easier to leave a highly coupled class undisturbed than to attempt to remove it. In other words, the disadvantages associated with its removal (i.e. side-effects, re-work and re-test) outweigh the disadvantages of simply leaving the class where it is.

The research in this chapter is motivated by a number of factors. First, we would always expect potentially problematic classes to be re-engineered by developers through techniques such as refactoring (Fowler, 1999); however, practical realities (limited time and resources) indicate that only when classes exhibit particularly bad 'smells' are they then dealt with (Fowler, 1999). The research in this chapter explores the characteristics of classes removed from a system, research that has not been touched on in any previous work that we know of. Throughout, we interpret the term 'removed' to mean that either a class has been:

a) Decomposed to form one or more newly named classes,

b) Moved to a different package and renamed or

c) Simply removed from the system because it is moribund.

Second, there is no prior study that we know of which suggests large classes with high coupling are removed any more or less frequently than small, low-coupled classes. Large classes may be a maintenance problem and hence candidates may be decomposed. On the other hand, however, small classes are more portable (and hence can be moved more easily). Finally, while there has been some work on finding the optimal size of class (El Emam, 2001), very little empirical research has investigated whether through analysis of removed classes, there is a coupling level beyond which action by the developer is usually triggered. The research described in this chapter relates to areas of software evolution, coupling metrics and the use of open-source software (Dinh-Trong and Bieman, 2004; Ferenc, 2004). In terms of software evolution, the basis for many past evolutionary studies has been provided by the laws proposed by Belady and Lehman (1976). Evolution has also been the focus in simulation studies (Smith et al., 2006). In terms of coupling, a framework for its measurement was introduced (Briand et al., 1999c); variations for different programming styles have also been proposed (Bartsch and Harrison, 2006a). Li and Henry (1993) verify that maintenance and tracing become more difficult with extreme coupling in the system. Chidamber and Kemerer (1994) proposed six OO metrics, amongst which were the Response for a Class and Coupling Between Objects coupling metrics. Finally, this study contributes to an empirical body of knowledge on coupling and longitudinal analysis of which more studies have been recommended (Kemerer and Slaughter, 1999a; 1999b).

5.3 Study Details

In this study, the main aim is to study the removed classes in an OSS. These classes are investigated by comparing the coupling they contain to other classes in the same package. Moreover, the size of these classes is considered together with the changes taking place in them before they are removed. These terms are investigated through three research questions using four OSS over several versions. Five coupling metrics are collected for each version for these systems.

5.3.1 Systems under Study

Four systems were used as a basis of our study. These systems are presented in Section 3.4.1; however, a brief description of them is as follows:

1) Jasmin. Jasmin is a Java assembler which takes ASCII descriptions of Java classes and converts them into binary Java .class files suitable for loading into a Java Virtual Machine. The system comprises 5 versions.

2) DjVu. DjVu is a Java system provides an applet and desktop viewer Java virtual machine. The system comprises 8 versions.

3) pBeans. pBeans is a Java system provides automatic object/relational mapping (ORM) of Java objects to database tables. The system comprises 10 versions.

4) Asterisk. The Asterisk Java system consists of a set of Java classes that allow the user to easily build Java applications that interact with an Asterisk PBX Server. It supports the FastAGI protocol and the Manager API. This system includes 6 versions.

5.3.2 Data Collected

OO metrics usually capture properties of OO systems such as cohesion, inheritance, encapsulation, polymorphism, size or coupling (Fenton and Pfleeger, 2002). For this study, the JHawk tool was used to collect five coupling metrics for each of the four systems (as described in Section 3.4.2). These metrics are:

1) Message Passing Coupling (MPC). The number of messages passed among objects of a class.

2) PACK. The number of the packages imported.

3) Number of EXTernal methods called (EXT). The more external methods that a class calls, the more tightly bound that class is to other classes.

4) Fan IN (FIN). The FIN of a function is the number of unique functions that call the function.

5) Fan OUT (FOUT). The FOUT counts the number of distinct non-inheritance related class hierarchies on which a class depends.

We also collected, for each removed class, the total number of methods (private, protected and public) and the lines of code (LOC) in each class as size measures.

5.3.3 The Research Questions

The study comprises three research questions (RQ1, RQ2 and RQ3), stated as follows:

- **RQ1:** Do removed classes contain significantly more or less coupling than other classes in the same package? This question is based on the belief that removed classes will tend to contain relatively small amounts of coupling when compared with other classes in the same package. We take the median coupling values of each metric within each package as a basis for our comparison. The median represents the mid-point of all values for that metric. All values below the median will be relatively ‘low’ values and values above, relatively ‘high’ values by comparison.

- **RQ2:** Are removed classes significantly ‘larger’ than other classes in the same package? This question is based on the belief that removed classes will tend to be small (in terms of their number of methods and LOC) when compared with other classes in the same package. Again, we take the median value for methods and LOC as a basis of our comparison.

- **RQ3:** Do removed classes tend to be modified significantly before they are removed? This question is based on the belief that classes which are modified significantly through versions of the systems studied are more likely to be removed because they cause frequent maintenance problems in the system.

5.4 Data Analysis

In order to assess our research questions, we collected the five coupling metrics, the number of methods (NOM) and LOC for the four systems. We then calculated the median for each variable. Subsequently, we presented the differences between the variables and their median values in tables. We calculated the differences for the five coupling metrics in order to assess the first research question, and the differences for the NOM and LOC to assess the second. For the third research question, we calculated the changes in the five coupling metrics for the removed classes over the course of the versions studied (prior to being removed) for each of the four studied systems. We assess each question on all the four systems separately.

5.4.1 Research Question 1 (RQ1)

Table 5.1 shows the name of removed classes, the name of the packages that they were removed from, the number of version in which the classes were removed and values for the five coupling metrics. These values are expressed as the real values for the metrics minus the median for that package and in the version where the class was removed. The median metric value for the package and for that version of the system is shown in brackets after each value in each case; if classes are removed in different versions, the median values for that particular version are shown. The values for the coupling metrics are plus or minus according to the difference between the real value of the metric and the median value. If the metric value is more than the median, then the value in the table is plus, and if the metric value is less than the median, then the value in the table is minus. For example, the MPC value for class *StackMapAttr* was 25 greater than the median value of 6 for that package (i.e. it had value 31). Equally, the MPC for class **Signed_num_token** was 4.5 less than the median MPC of 4.5 in that package (i.e. zero). Since both *StackMapAttr* and *StackMapFrame* classes were removed in the same version they share the same set of median values given in the first row (this is not always the case).

We have also highlighted the fact that classes are taken from different packages by alternating *italicized* class values with bold un-italicized values. Consequently, the first two classes in Table 5.1 are from one package *jas* and the third class from a different package *jasmin*. We note also that the values in brackets represent the median for the *whole* package and therefore apply to *all* similar rows below it in the same table.

For the Jasmin system, the three removed classes were all found in the fourth version (out of five). The first two removed classes are higher than the median for the coupling metrics. For the third class, all but one of the same metrics are below the median. Clearly, for this system, coupling exceeds the median in the majority of cases. This is more noticeable in Figure 5.1, where the differences for the coupling metrics from the median are presented for each removed class (We refer to the class by the number of the row that presents it in the table).

Table 5.1 Removed classes compared to median (Jasmin)

Removed Class	Package	In	MPC	EXT	PACK	FOUT	FIN
StackMapAttr	*Jas*	*V4*	*25 (6)*	*13(5)*	*2 (1)*	*6 (0)*	*0 (0)*
StackMapFrame	*Jas*	*V4*	*15*	*7*	*2*	*2*	*3*
Signed_num_token	**jasmin**	**V4**	**-4.5 (4.5)**	**-3.5 (3.5)**	**-0.5 (1.5)**	**-1.5 (1.5)**	**0 (0)**

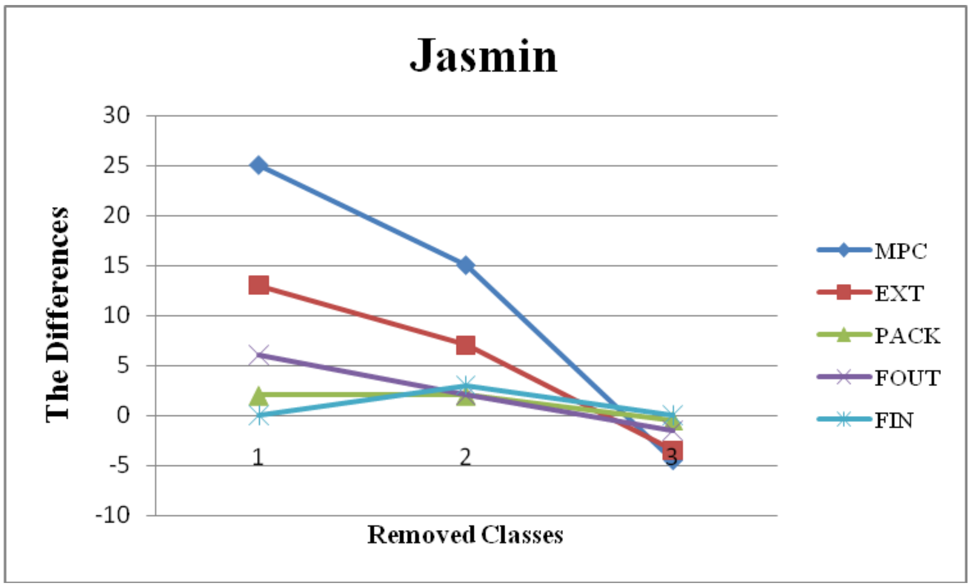

Figure 5.1 Coupling metrics in the removed classes compared to median (Jasmin)

For DjVu (Table 5.2), a number of different patterns emerge. First, it seems that when a class was removed, it tended to have relatively low (i.e. minus) MPC, EXT and PACK values compared with other classes (given by the median). For example, seven of the twelve removed classes contained values of MPC significantly less than the median; five of the twelve removed classes contained 10 or less EXT values than the median. The same trend applies to the PACK metric. (It is relatively easy to remove a class that is lowly-coupled in terms of message passing and external calls.) Equally, with the exception of one class, the values of FOUT for this system are either 0 or negative. This is not always the case for FIN, suggesting a difference in emphasis between these two metrics when removing classes. A class with a higher FOUT than FIN is, in theory, easier to remove because it has fewer incoming dependencies than outgoing. Interestingly, only in four of the twelve cases does this occur in Table 5.2. Nonetheless, the values of FIN and FOUT are generally low; for two of the packages every FOUT value of removed classes is less than or identical to the median value. Also of note are the exceptionally low values of MPC and EXT for the third package (each of the three classes in this package was removed in different versions because they each have their own set of median values). Overall, of the sixty values for all metrics in the table, 33 were negative and 7 equivalent to the median (value of zero in the table).

Table 5.2 Removed classes compared to median (DjVu)

Removed Class	Package	In	MPC	EXT	PACK	FOUT	FIN
GMapRect	*Djvu*	*V3*	*12 (20)*	*3 (15)*	*0 (1)*	*-1 (5)*	*4 (10)*
GRectMapper	*Djvu*	*V3*	*7*	*1*	*-1*	*-5*	*-10*
LibRect	*Djvu*	*V3*	*-15*	*-10*	*1*	*-5*	*4*
Annotation	*Djvu*	*V3*	*-20*	*-15*	*-1*	*-5*	*0*
ByteVector	**Djvu**	**V7**	**5.5 (20)**	**2(15)**	**0 (2)**	**0 (5)**	**-2.5 (10.5)**
DataPool$ CachedInputStream	**Djvu**	**V7**	**1.5**	**1**	**-2**	**0**	**-10.5**
IFFContext	**Djvu**	**V7**	**-9.5**	**-8**	**-1**	**-3**	**-7.5**
GMapOval	*djvu.anno*	*V3*	*-17 (27)*	*-7 (17)*	*1 (1)*	*0 (4)*	*3 (0)*
GMapPoly	*djvu.anno*	*V3*	*51*	*20*	*1*	*1*	*3*
BoundImage	**Djvubean**	**V3**	**-41(42)**	**-27.5 (28.5)**	**0 (5)**	**-7.5 (7.5)**	**1 (3)**
DjVuBean$ HyperlinkListener	*Djvubean*	*V5*	*-46 (59)*	*-30 (43)*	*-5 (5)*	*-7 (11)*	*-5 (5)*
SimpleArea	**Djvubean**	**V6**	**-50 (75)**	**-34 (48)**	**-4 (6)**	**-10.5 (12)**	**0.5 (4.5)**

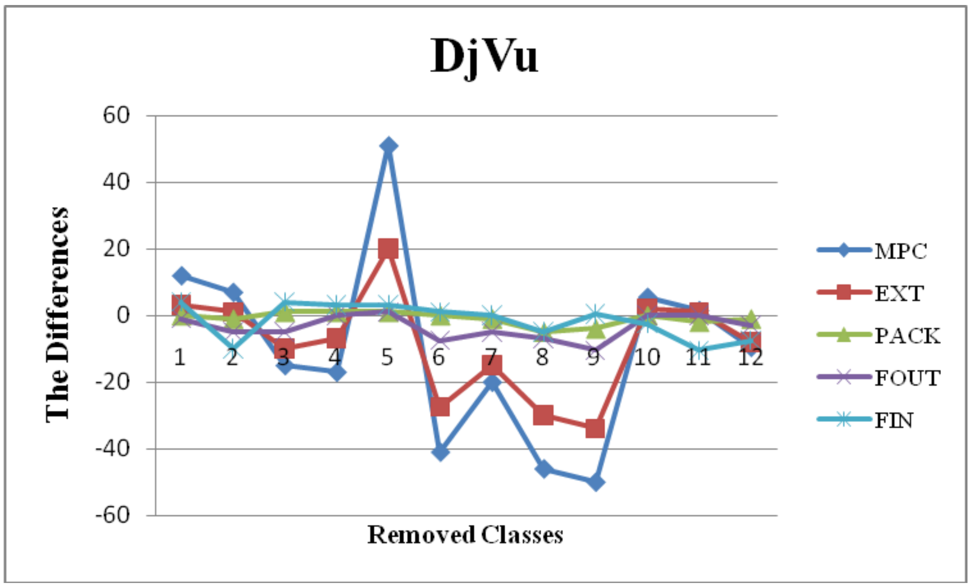

Figure 5.2 Coupling metrics in the removed classes compared to median (DjVu)

Table 5.3 shows the coupling metrics for the pBeans system expressed as values plus or minus the median. Eleven classes were removed from three different packages. In common with the DjVu system, the FIN values seem to be low compared with the median value and in this case, so too the FOUT. The most notable feature of Table 5.3 is the fact that all six classes removed from the second package (bolded) relate explicitly to databases. Moreover, the MPC and EXT values are exceptionally high for these classes. There is a reasonable explanation for this feature. Database classes are more likely to be used extensively by other classes and that could explain the high MPC and EXT values (the PACK values for the same classes are relatively low). It might be the case that these six classes may not have been removed necessarily, but simply ‘moved’ all together as part of an ‘Extract Package’ refactoring to re-locate database classes where they are most needed (Fowler, 1999). It is interesting that not all of the same six classes had low FIN and FOUT values, suggesting that only some forms of coupling may be relevant or considered by a developer when deciding on class removal. Of the 55 values in Table 5.3, only 21 values were negative. The majority of positive values were accounted for by the database classes.

Table 5.3 Removed classes compared to median (pBeans)

Removed Class	Package	In	MPC	EXT	PACK	FOUT	FIN
ObjectClass	*pbeans*	*V8*	*-2 (2)*	*-2 (2)*	*0 (0)*	*-1.5 (1.5)*	*1 (5)*
ObjectClass_StoreInfo	*pbeans*	*V8*	*2*	*2*	*2*	*0.5*	*-5*
PersistentMap Entry_StoreInfo	*pbeans*	*V8*	*-1*	*-1*	*0*	*-0.5*	*-5*
PersistentMap_StoreInfo	*pbeans*	*V8*	*-1*	*-1*	*0*	*-1.5*	*-5*
HsqlDatabase	**data**	**V8**	**33.5 (0.5)**	**27.5 (0.5)**	**4.5 (0.5)**	**15.5 (0.5)**	**-1.5 (1.5)**
HsqlDatabase$ UpperCaseMap	**data**	**V8**	**3.5**	**2.5**	**-0.5**	**0.5**	**-1.5**
PostgreSQLDatabase	**data**	**V8**	**36.5**	**32.5**	**3.5**	**17.5**	**-1.5**
PostgreSQLDatabase$ LowerCaseMap	**data**	**V8**	**3.5**	**2.5**	**-0.5**	**0.5**	**-1.5**
MySQLDatabase	**data**	**V8**	**9.5**	**6.5**	**2.5**	**3.5**	**-1.5**
SQLServerDatabase	**data**	**V8**	**18.5**	**14.5**	**3.5**	**10.5**	**-1.5**
InitFilter	*servlet*	*V8*	*-1(10)*	*0 (8)*	*3 (3)*	*1(5)*	*0 (2)*

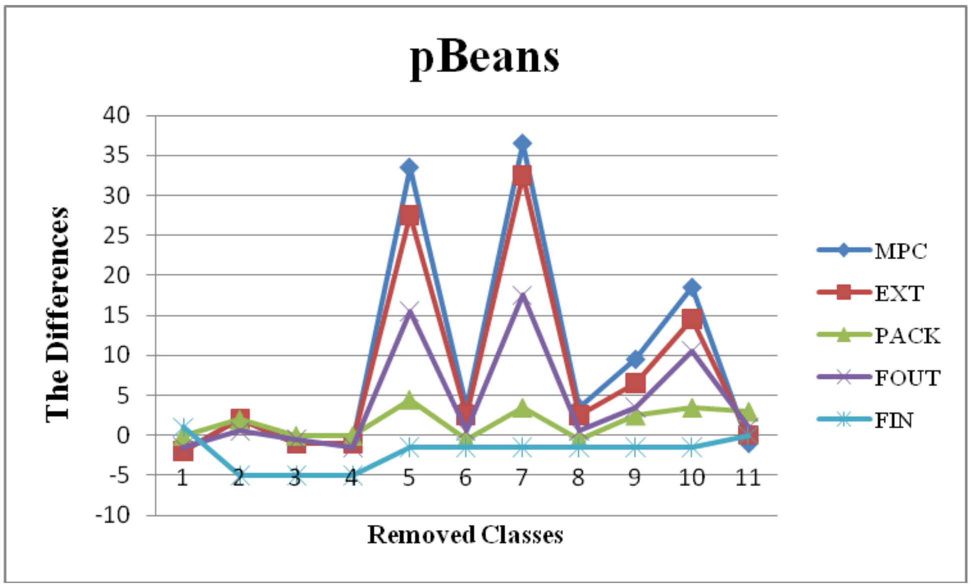

Figure 5.3 Coupling metrics in the removed classes compared to median (pBeans)

For the Asterisk system (Table 5.4), eight classes were removed from four packages. The Asterisk system exhibits a similar pattern to the DjVu system in terms of the FIN and FOUT values, the majority of which were either zero or negative when compared with the median. In keeping with the pBeans system, the MPC and EXT values for removed classes are quite large in many cases.

Consider, for example, the classes **ReplyBuilderImpl**, **ReplyBuilderImplTest**, **RequestBuilderImpl** and **RequestBuilderImplTest** – all of which have high MPC and EXT values. Finally, the two classes **ServerSocketFacadeImpl** and **SocketConnectionFacadeImpl** are related to patterns and, in particular, the facade pattern (evidence by Bieman et al. (2003) suggests that pattern classes are more susceptible to change than non-pattern based classes). The same phenomenon of moving related classes such as those for the database classes of pBeans may apply here.

Table 5.4 Removed classes compared to median (Asterisk)

Removed Class	Package	In	MPC	EXT	PACK	FOUT	FIN
ReplyBuilder	*fastagi*	*V2*	*0 (0)*	*0 (0)*	*1.5 (0.5)*	*0 (0)*	*-1 (1)*
RequestBuilder	*fastagi*	*V2*	*0*	*0*	*0.5*	*0*	*-1*
ReplyBuilderImpl	**impl**	**V2**	**17 (12)**	**15 (10)**	**-2 (5)**	**9 (7)**	**0 (0)**
ReplyBuilderImplTest	**impl**	**V2**	**36**	**3**	**-3**	**-1**	**0**
Request BuilderImpl	**impl**	**V2**	**55**	**41**	**5**	**7**	**0**
RequestBuilder ImplTest	**impl**	**V2**	**101**	**19**	**-2**	**0**	**0**
CommonsLoggingLog	*util*	*V2*	*5 (2)*	*3 (2)*	*0.5 (0.5)*	*-1(2)*	*0 (0)*
NullLog	*util*	*V2*	*-2*	*-2*	*-0.5*	*-2*	*0*
ServerSocket FacadeImpl	**asterisk.io**	**V4**	**4 (0)**	**4 (0)**	**2.5 (1.5)**	**2 (0)**	**-7 (7)**
SocketConnection FacadeImpl	**asterisk.io**	**V4**	**18**	**12**	**4.5**	**1**	**-7**
Util	*manager*	*V4*	*5.5 (2.5)*	*4 (2)*	*-2 (2)*	*0 (1)*	*7 (0)*

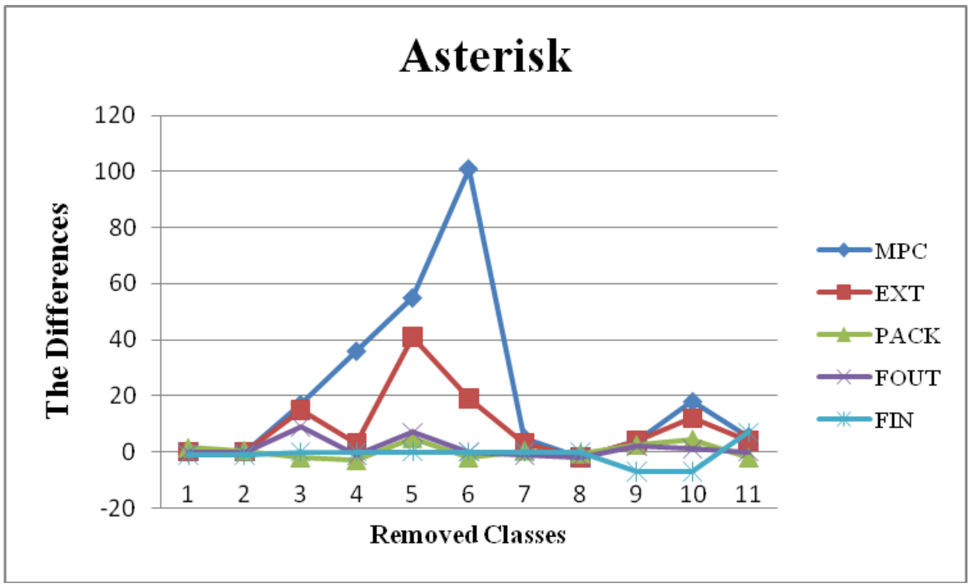

Figure 5.4 Coupling metrics in the removed classes compared to median (Asterisk)

In response to RQ1, we suggest that FIN and FOUT coupling may be a strong determinant of whether a class is removed – low values of each may help the removal of a class; equally, high amounts of MPC and EXT may actually be one stimulus for moving a class. However, the key driver for removing classes as noted for classes in pBeans and Asterisk may be the need to remove related classes to a more convenient location.

5.4.2 Research Question 2 (RQ2)

Research question 2 attempts to answer the question whether removed classes were significantly ‘larger’ than other classes in the same package? We determined the size of the class by the number of methods NOM and the number of LOC for this class.

In order to answer the research question, we compare the size of the removed classes by taking the median value for NOM and LOC as a basis of this comparison. Table 5.5 shows the name of removed classes for each system, and values for the NOM and LOC. These values are expressed as the real values minus the median for that package and in the version where the class was removed; these values are minus or plus depending on whether the real values are less or more than the median, correspondingly.

Table 5.5 NOM and LOC compared to the median for the four systems

System	Removed class	NOM	LOC
Jasmin	StackMapAttr	1	14.5
	StackMapFrame	4	19.5
	Signed_num_token	-2	-21
pBeans	GMapRect	32.5	113
	GRectMapper	9.5	93
	LibRect	-11.5	-41
	GMapOval	0	-77
	GMapPoly	26	116
	BoundImage	-14	-59
	Annotation	-0.5	-65
	DjVuBean$HyperlinkListener	-12	-79
	SimpleArea	-2	-185.5
	ByteVector	0	5

	DataPool$CachedInputStream	1	-31
	IFFContext	-5	-47
	ObjectClass	1	-5
SQL	ObjectClass_StoreInfo	-3	-8
	PersistentMapEntry_StoreInfo	-3	-14
	PersistentMap_StoreInfo	-3	-14
	HsqlDatabase	5	33.5
	HsqlDatabase$UpperCaseMap	-3	-9.5
	PostgreSQLDatabase	9	47.5
	PostgreSQLDatabase$LowerCaseMap	-3	-9.5
	MySQLDatabase	1	-0.5
	SQLServerDatabase	3	26.5
	InitFilter	1	11
	IndexNodeFile	-1	-12.5
Asterisk	ReplyBuilder	-1	0
	RequestBuilder	-1	-1
	ReplyBuilderImpl	-3	9
	ReplyBuilderImplTest	5	16
	RequestBuilderImpl	2	83
	RequestBuilderImplTest	11	94
	CommonsLoggingLog	4.5	5.5
	NullLog	4.5	-2.5
	ServerSocketFacadeImpl	-2.5	4
	SocketConnectionFacadeImpl	1.5	17
	Util	1	19.5

In order to study the NOM and LOC separately, we used the line chart to present each of their values in a different figure. Figure 5.5 shows the values of LOC for classes for each system and Figure 5.6 shows the NOM for the same four systems. The 'zero' vertical axis represents the median value of NOM and LOC in the four systems. Hence, plotted values represent NOM and LOC values above (plus) or below (minus) the median.

Figure 5.5 seems to show that a similar number of the 37 removed classes had LOC values below the median as above it; in fact 21 of the 37 were either zero or above and therefore 16 were below the median.

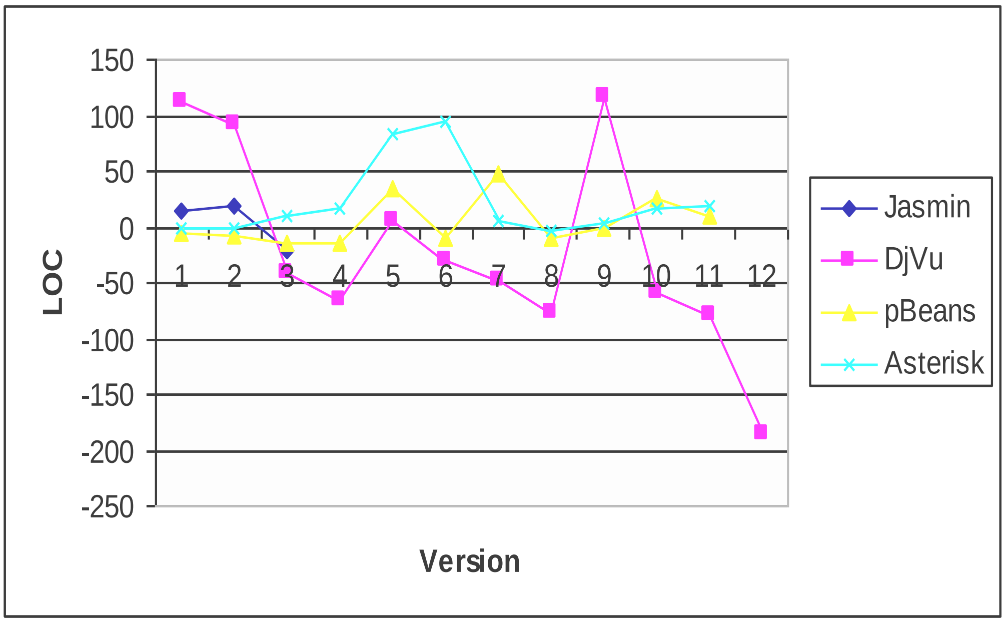

Figure 5.5 LOC in removed classes

Figure 5.6 shows a similar pattern to Figure 5.5. Of the 37 values, 19 were zero or above (18 value were therefore below). In both figures, the DjVu system seems to be the system where both large and small classes were removed from the system (given by the erratic peaks). These results suggest that size, both in terms of NOM or LOC, seemed to have little bearing on the choice of removal of a class. A similar effect appears to take place for the pBeans and Asterisk systems, but to a lesser extent. For the Asterisk system, the peak in NOM and LOC coincides with the high values for the second package in Table 5.4. This implies that for this system, removed classes were both highly coupled and relatively large.

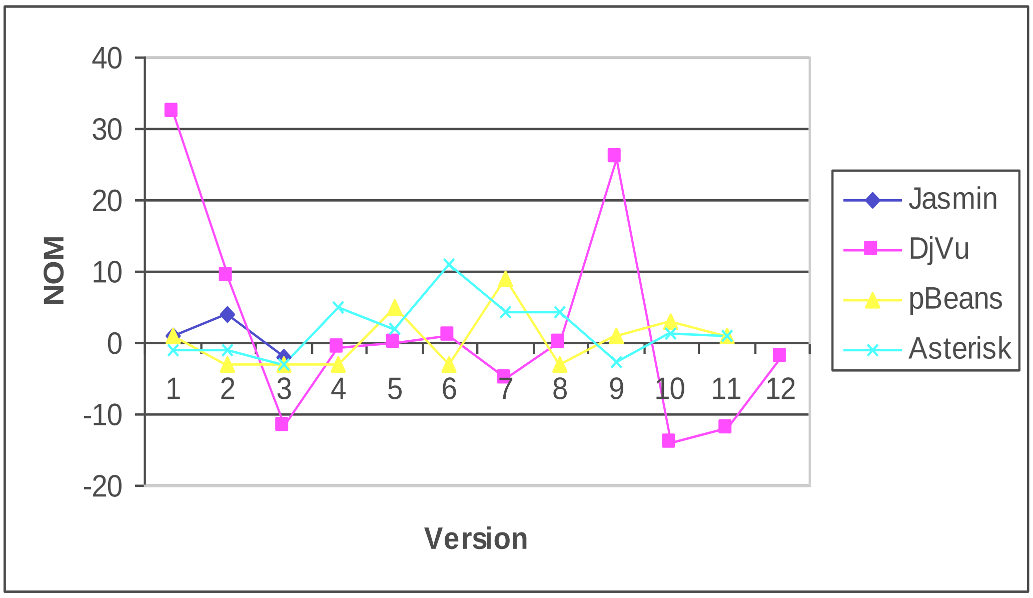

Figure 5.6 NOM in removed classes

Based on the evidence presented, and in response to RQ2, size does not seem to be a key determinant in the removal of a class. Both small and large classes were removed (coupling may be a far greater determinant). This result is supported by Counsell (2008) where size was found to be a poor predictor of OO cohesion; coupling was a far better determinant.

5.4.3 Research Question 3 (RQ3)

Research question three aims to answer the question whether removed classes were also the subject of significant changes over the course of the versions studied (prior to being removed). For the Jasmin system, Table 5.6 shows the number of classes of the set of removed classes that were the subject of changes during the five versions of the system studied. As before, the values in the table are relative to the median. For example, two of the three removed classes in the Jasmin system had had changes applied to them. Class *StackMapAttr* was removed ‘In’ version 4 and had changes applied to it between version 2 and 3. The same happened to class *StackMapFrame*. It was removed in version 4 and had changes applied to it between version 2 and 3. Both of these classes were removed from the jas package. However, for the third removed class **Signed_num_token**, no change had been applied to it over the course of the versions prior to its removal, so it does not appear in Table 5.6.

Table 5.6 Changes for removed classes (Jasmin)

Removed classes	In	Changes	NOM	LOC	FOUT	FIN
StackMapAttr	*V4*	*V2-V3*	*0*	*0*	*-1*	*0*
StackMapFrame	*V4*	*V2-V3*	*2*	*6*	*-1*	*-5*

For the DjVu system (Table 5.7), five out of the twelve classes were changed and these changes occurred between the second and the third versions in every case. However, these classes were not removed directly after those changes, they were removed later in the fifth, sixth, and seventh versions. These classes were removed from three different packages.

Table 5.7 Changes for removed classes (DjVu)

Removed classes	In	Changes	NOM	LOC	FOUT	FIN
ByteVector	*V7*	*V2-V3*	*0*	*-4*	*-3*	*-1*
DataPool$CachedInputStream	*V7*	*V2-V3*	*0*	*0*	*0*	*0*
IFFContext	*V7*	*V2-V3*	*0*	*0*	*0*	*-1*
DjVuBean$HyperlinkListener	**V5**	**V2-V3**	**0**	**2**	**0**	**0**
SimpleArea	*V6*	*V2-V3*	*0*	*0*	*0*	*-6*

For the pBeans system (Table 5.8), there were changes in just three classes out of eleven. However, most of these changes were in the first three versions and they were all removed in the eighth version. The class **MySQLDatabase** was changed twice during the period studied and the class **SQLServerDatabase** was modified three times over the course of the versions studied (they thus have two and three entries in Table 5.8, respectively). These three classes were removed from the same package “data”.

Table 5.8 Changes for removed classes (pBeans)

Removed classes	In	Changes	NOM	LOC	FOUT	FIN
PostgreSQLDatabase	**V8**	**V3-V4**	**2**	**8**	**1**	**0**
MySQLDatabase	**V8**	**V1-V2**	**1**	**6**	**1**	**0**
MySQLDatabase	**V8**	**V6-V7**	**0**	**3**	**1**	**0**
SQLServerDatabase	**V8**	**V1-V2**	**1**	**13**	**4**	**0**
SQLServerDatabase	**V8**	**V2-V3**	**0**	**6**	**1**	**0**
SQLServerDatabase	**V8**	**V3-V4**	**2**	**8**	**1**	**0**

Finally for the Asterisk system (Table 5.9), there were changes in just three classes out of eleven. However, these changes were in the first two versions and it was not until the fourth version that they were removed.

Table 5.9 Changes for removed classes (Asterisk)

Removed classes	In	Changes	NOM	LOC	FOUT	FIN
SocketConnectionFacadeImpl	**V4**	**V2-V3**	**1**	**2**	**0**	**0**
Util	*V4*	*V1-V2*	*1*	*6*	*0*	*1*
Util	*V4*	*V2-V3*	*0*	*0*	*0*	*2*

The conclusion we can draw in response to RQ3 is that first, removed classes are not necessarily changed significantly prior to their removal for the systems analysed. Second, that removal of the classes took place at a later date to that of change in all cases investigated. This was a surprising result to emerge from the analysis. Finally, we note that for all the changed systems in Tables 5.6-5.9, the FIN and FOUT values are small even when compared with the other FIN and FOUT values in Tables 5.1-5.4.

5.5 Study Validity

First, we have identified removed classes but could not say whether these classes were simply moved to a different package and renamed (we would expect most removed classes to be decomposed and for the subsequent classes to be renamed). To counter this threat to validity, we did search for classes with different names but which had identical compositions to those removed classes, but found very little evidence to suggest that classes are actually moved and renamed (i.e. they tend to be decomposed or simply removed from the system). Finally, we have focused on coupling and size as the basis of our analysis. We could have used many other features of classes as a basis; for example, their cohesion or their position in the inheritance hierarchy (Cartwright and Shepperd, 2000). We leave such analyses for future work, however.

5.6 Summary

In this chapter, we investigated removed classes in four Java systems. Five coupling metrics were collected from four Java open-source systems using the JHawk tool. The study investigated three research questions. First, we investigated whether the extent of coupling influenced the removal of classes from a system. We found that the FIN and FOUT metrics tended to be relatively small for removed classes. Moreover, that imported functionality (packages) and external calls play a role in certain cases (we found evidence of movement of database classes with high levels of message passing and external references). Second, we explored whether size was an influence on removed classes. We found little evidence that size did influence that choice. Finally, the expectation that removed classes were changed significantly before being removed was ill-founded; changes for most of the classes were made in early versions and removed relatively later on.

In the next chapter, coupling will be empirically investigated. Five coupling metrics will be explored in five Java systems. The coupling will be examined in terms of their relationships with the version release times and code warnings.

CHAPTER 6. EMPIRICAL OBSERVATIONS ON COUPLING AND CODE WARNINGS

6.1 Introduction

In the previous chapter, an empirical study concerning removed classes in Java OSSs was undertaken. In the study, removed classes in four OSSs were investigated through three related research questions. First, does the amount of coupling influence the choice of removed class? Second, does class size play a role in that choice? Finally, is there a relationship between the frequency with which a class is changed and its point of removal from a system?

One question that is pertinent to ask about coupling based on the features extracted in Chapter 4 and 5 is the extent to which it might cause problems in code. In other words, does excessive coupling cause faults to be invested in code or at best induce a coding style that naturally harbours faults? In this chapter, we explore this aspect of coupling. Our investigation considered coupling in five Java systems using coupling metrics, version release times and code warnings. We collected five coupling metrics, class data and version release times from the systems using the JHawk tool and used code warnings extracted using the FindBugs tool to determine the relationships between coupling, those warnings and the time interval between versions.

Results found that addition of coupling may have beneficial effects on a system. It also seems that addition of coupling in new functionality through packages could result in fewer warnings than adding functionality to existing code. Finally, there appears to be a coupling trade-off between coupling types – in particular that between the uses of coupling through imported packages and the introduction of ‘internal-to-the-package’ coupling.

The remainder of the chapter is organised as follows. In the next section, we present the motivation for the study and related work. In Section 6.3, we provide details of the systems studied, the tools used, the data collected and the research questions. We then present the data analysis including the role that code warnings played (Section 6.4). We then discuss a number of issues raised by the study in Section 6.5 before concluding in Section 6.6. We note that the research in this chapter was first published by Mubarak et al. (2008b).

6.2 Study Motivation

Coupling, whether in the procedural or OO paradigm, has often been related to the propensity for faults in software (Briand et al, 1997; Briand et al., 1998). It is generally accepted in the OO software engineering community that extreme coupling between classes produces a level of complication that makes problems with subsequent maintenance and possibly guides to the seeding of (further) faults. The research in this paper is motivated by a number of factors. Firstly, the research in Chapter 4 has shown that frenetic bursts of refactoring activity after specific releases of a system, suggesting that this activity is in response to a wide range of ‘regular’ (i.e. non-refactoring) changes to the system under consideration. There is a strong link between refactoring and the need to reduce coupling and it is thus a natural extension to the research in this earlier work to explore those regular changes and, moreover, their link with refactoring. Secondly, while there have been many studies of evolving systems, the time frame between releases is often ignored, and each version release is considered as occurring at an equal time interval from the last. However, analysis of relative change may reveal significant facets of the maintenance activity that, in particular, have a relationship with trends in fault propensity.

While there has been a large amount of research into evolutionary trends in systems in the past (Belady and Lehman, 1976; Bieman et al., 2003; Girba and Ducasse, 2006; Lehman, 1980; Mens et al., 2004), a number of research questions remain mainly unaddressed. Firstly, releases of a system can arise at very different time intervals, and change activity can be motivated by a number of factors. For example, it is possible for two sequential versions of a system to be released on the same day because of a requirements fault in the primary release. In other cases, time intervals of over a year between version releases are common. The research question that naturally arises is: what trends in change activity can we observe if we factor in the different time periods between releases of a system? Secondly, if we can observe that there are these concerted ‘bursts’ of maintenance activity in which increased coupling will inevitably feature, then how can potential code ‘warnings’ assist and inform our understanding during or after those bursts? A high proportion of change activity must inevitably have an effect on potential fault patterns.

6.3 Preliminaries

6.3.1 Systems under Study

Five systems were used in order to investigate the research questions. Three of these systems were used in the study conducted in Chapter 5 (Jasmin, pBeans and DjVu). The five systems are presented in Section 3.4.1; however, a brief description of them is as follows:

1. *Jasmin.* Jasmin is a Java assembler which takes ASCII descriptions of Java classes and converts them into binary Java .class files suitable for loading into a Java Virtual Machine. The system comprises 5 versions.
2. *DjVu.* DjVu is a Java system that provides an applet and desktop viewer Java virtual machine. The system comprises 8 versions.
3. *pBeans.* pBeans is a Java system which gives automatic object/relational mapping (ORM) of Java objects to database tables. The system comprises 10 versions.
4. *SmallSQL.* Small SQL is a Java DBMS for Java desktop applications. It has a JDBC 3.0 interface and offers many ANSI SQL 92 and ANSI SQL 99 features. The system comprises 8 versions.
5. *JWNL.* JWNL is a Java API for accessing the WordNet relational dictionary. WordNet is widely used for developing NLP applications and allows developers to use Java for building NLP applications. The system comprises 5 versions.

6.3.2 Tools Used

Two software tools were used for our analysis. Firstly, the JHawk (2008) tool was used to collect the five coupling metrics and the class data (number of classes and methods). The FindBugs (2008) tool was used to collect the warnings for each version of the systems. The FindBugs tool analyses the Java byte code for common potential fault patterns and issues those warnings decomposed into six categories.

1. *Bad Practice (BP)*: "Violations of recommended and essential coding practice. Examples include hash code and equals problems, serializable problems/misuse of finalize."
2. *Correctness (CORR)*: "An apparent coding mistake resulting in code that was probably not what the developer intended." For example, method ignores return value/double assignment of field.
3. *Malicious Code Vulnerability (MCV)*: State where internal information is changed or exposed. Examples include that a mutable static field could be changed by malicious code or by accident from another package.
4. *Multi-threaded Correctness (MTC)*: A potential fault due to careless housekeeping of threads. Examples include a method that does not release a lock on all paths, and field not guarded against concurrent access.
5. *Performance (PER)*: Code written in such a way that would detract from the efficiency of the system. Examples include a private method never being called, an unread or unused field, and inappropriate use of String.
6. *Questionable (Dodgy) Practice (DODGY)*: "Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, unconfirmed casts and redundant null check of value known to be null."

6.3.3 Data Collected

For each of the systems, we collected five independent and orthogonal coupling metrics and one time based metric (as described in Section 3.4.2). The first four metrics were used also in the study conducted in Chapter 5 (Section 5.3.3).

1. *Message Passing Coupling (MPC)*: The number of messages passed among objects of a class.
2. *PACK.* The number of imported packages.
3. *Fan IN (FIN).* The FIN of a function is the number of unique functions that call the function.
4. *Fan OUT (FOUT).* The FOUT counts the number of distinct non-inheritance related class hierarchies on which a class depends.
5. *Response for a Class (RFC).* This metric is the same as that defined by Chidamber and Kemerer (1994) and measures the response set of a class. The RFC is defined as the set of methods that can potentially be executed in response to a message received by an object of that class.
6. The time intervals between each version release.

6.3.4 Research Questions

The study comprises two research questions (RQ1 and RQ2), stated as follows:

- **RQ1:** What trends in change activity can we observe if we factor in the different time periods between releases of a system? This question is based on the belief that the time interval between two released versions will affect the changes in a system if we put it under consideration. Sometimes the time interval between the version releases can be days, while sometimes it can be months.
- **RQ2:** How can potential code 'warnings' help and inform our expectations of the changes in a system activity? This research question is based on the fact that a high fraction of change activity certainly has an effect on potential fault trends.

6.4 Data Analysis

For each of the following five systems, we present the coupling data, the warnings for each version and the time interval between versions. We will assess the two research questions at the same time for each of the five systems.

6.4.1 The Jasmin System

Table 6.1 shows the changes in each of the coupling metrics over the five versions of the Jasmin system. No new packages were introduced over the course of the five versions, but we observe significant changes in each of the coupling metrics particularly between releases version 2 and 3, before falling consistently afterwards. The number of added classes was relatively low, but the addition of 21 classes between versions 1 and 3 resulted in over 150 new methods being added. Between versions 4 and 5, there were small amounts of added coupling.

Table 6.6.1 Changes in coupling metrics for the Jasmin system

Jasmin	Interval	Packages	Classes	RFC	MPC	PACK	FOUT	FIN
V1-V2	401	0	10	202	194	13	29	30
V2-V3	63	0	11	317	330	25	42	31
V3-V4	45	0	-1	45	44	-1	11	11
V4-V5	140	0	0	1	5	0	6	2

One feature of the data was not a surprise - the ‘burst’ and then sudden fall in coupling activity was noted previously in Chapter 4, where refactoring effort for OSS followed a similar pattern of: bursts of maintenance activity followed almost immediately by bursts of refactoring activity.

Table 6.2 shows the warnings for each release of the Jasmin system. These warnings are presented in the six aforementioned categories. Figure 6.1 shows the total number of warnings for each release of the Jasmin system (upper graph) and the changes in the number of warnings (lower graph). It is remarkable that from version 2 to 3, there was actually a fall in the number of warnings for this system (and yet this was accompanied by a significant rise in coupling as we can see from Table 6.1).

Table 6.6.2 Warnings for Jasmin

Jasmin	BP	CORR	MCV	MTC	PER	DODGY	Total
V1	6	0	8	0	1	2	17
V2	6	0	8	0	18	7	39
V3	7	0	8	0	7	9	31
V4	8	0	10	0	7	12	37
V5	0	0	0	0	0	0	0

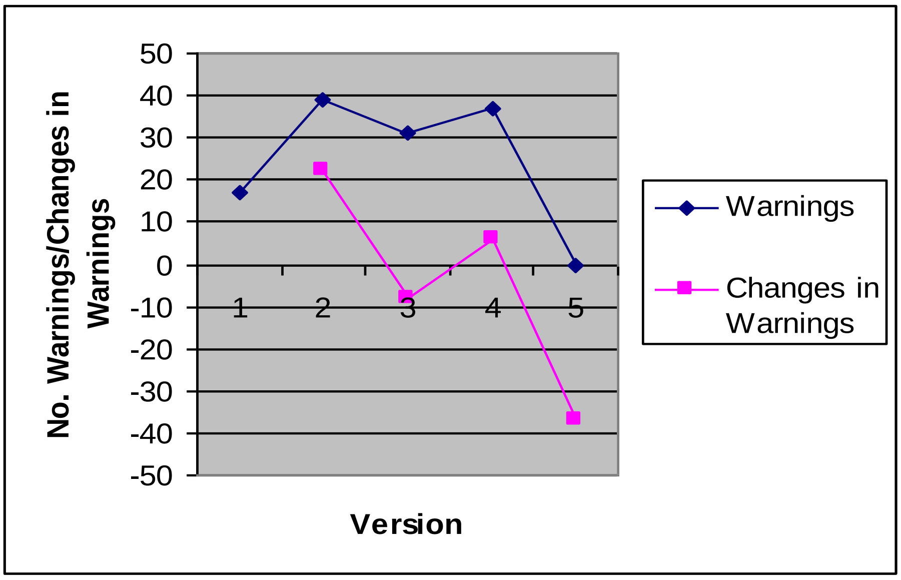

Figure 6.1 Total warnings for Jasmin

Inspection of the warning categories issued by FindBugs revealed that over 50% of the 39 warnings attributed to version 2 were found to be in the performance (PER) category. The majority of the warnings in this category relate to the need for additional method invocation to overcome inefficiencies associated with data manipulation (this would affect the values of RFC and MPC metrics in particular). In many instances, the developer is urged by FindBugs to heed this warning by *adding* method calls (coupling) to specific classes and, in some cases, to replace one method call with two. For example, the following complex construct replaces a single method call as a remedy to one warning related to improper array use: *myCollection.toArray(newFoo[myCollection.size()])*. In other words, the increase in coupling witnessed by the Jasmin system between version 2 and 3 may have been from necessity. More significantly and counter-intuitively, added coupling may actually have contributed to the decrease in warnings between those versions. We cannot therefore discount the possibility that increases in coupling may actually have beneficial effects in a system by eliminating potential inefficiencies. This was a surprising feature to emerge from our study.

The values in Table 6.1 and Figure 6.1 make the assumption that between each release of the Jasmin system, there is an equal length of physical time. Figure 6.2 shows the time intervals between each of the versions of Jasmin.

The significant increase in coupling between version 2 and 3 is placed in its proper context when we consider that there were 401 days between version 1 and 2, yet only 63 days between version 2 and 3. We could suggest that a key motivation for the burst of increased coupling between version 2 and 3 (and the added coupling therein) may have been simply to improve the performance of the system. This may also explain the minimal changes in coupling and the consequent drop in warnings thereafter.

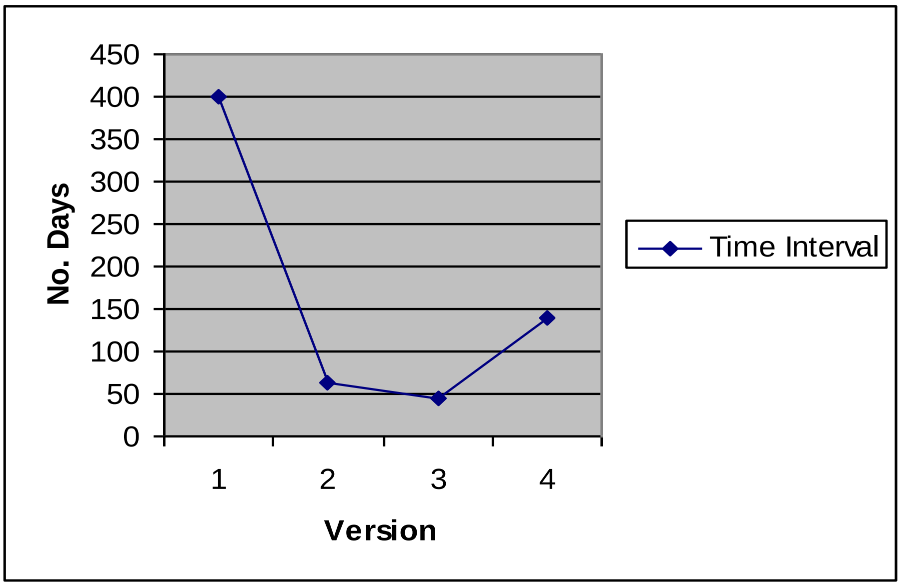

Figure 6.2 Time interval between versions

To conclude, contrary to what we would expect, added coupling may have a positive rather than detrimental effect on a system. This result also points to the possibility of adding 'good' coupling to a system as well as removing 'bad' coupling in a simultaneous operation.

6.4.2 The pBeans System

Table 6.3 shows the changes in each of the five coupling metrics over the ten versions of the pBeans system. A notable feature of the values in the table is the relatively low coupling activity between version 3 and 6. Thereafter, there is a significant increase in each of the metric values. This increase would appear to be due to the addition of 6 new packages over the course of versions 6-8. The only decrease in a metric value was attributed to FIN between versions 1 and 2.

Table 6.3 Changes in coupling metrics for the pBeans system

pBeans	Interval	Packages	RFC	MPC	PACK	FOUT	FIN
V1-V2	2	0	5	6	1	5	-2
V2-V3	5	0	92	102	8	27	34
V3-V4	2	0	16	14	0	1	0
V4-V5	17	0	2	3	0	0	3
V5-V6	35	0	44	39	5	18	1
V6-V7	297	1	113	89	25	49	32
V7-V8	727	5	604	607	34	208	190
V8-V9	3	0	15	23	0	4	4
V9-V10	26	0	18	28	0	3	3

In Table 6.4, the number of warnings for each of the releases of pBeans system are categorised in the same six groups. Figure 6.3 shows the total number of these warnings and seems to follow the pattern of the values in Table 6.3. Figure 6.4 shows the wide variation in times between each of the versions of the system. A surprising (and notable) feature of Figure 6.3 and Table 6.3 is the relatively low rise in warnings accompanying the large time interval after version 6, a period in which large amounts of coupling was added to the system. The rise in warnings between version 2 and 3 (Figure 6.3) is actually greater than that after version 6.

One explanation for this feature might be that adding new packages does not *per se* cause a corresponding rise in warnings. In other words, self-contained and encapsulated new packages tend to induce relatively few warnings. We thus suggest that there is a marked and distinct difference between adding coupling to those *existing* packages and the consequent effect this has on warnings when compared with the influence on warnings through the addition of *new* packages.

From a maintenance point of view, we would normally expect new code to create fewer ‘ripple’ effects (Black, 2001) and to generate fewer warnings than modification of existing code (because of the lower potential for lack of code comprehension and the possibility of side-effects).

Table 6.6.4 Warning for pBeans

pBeans	BP	CORR	MCV	MTC	PER	DODGY	Total
V1	146	15	14	38	33	51	297
V2	146	15	14	38	32	51	296
V3	204	30	23	39	40	70	406
V4	203	27	20	39	38	65	392
V5	204	30	23	39	40	70	406
V6	204	31	23	39	40	71	408
V7	211	31	30	39	41	73	425
V8	228	45	40	34	59	100	506
V9	229	45	40	34	59	104	511
V10	230	45	40	34	59	104	512

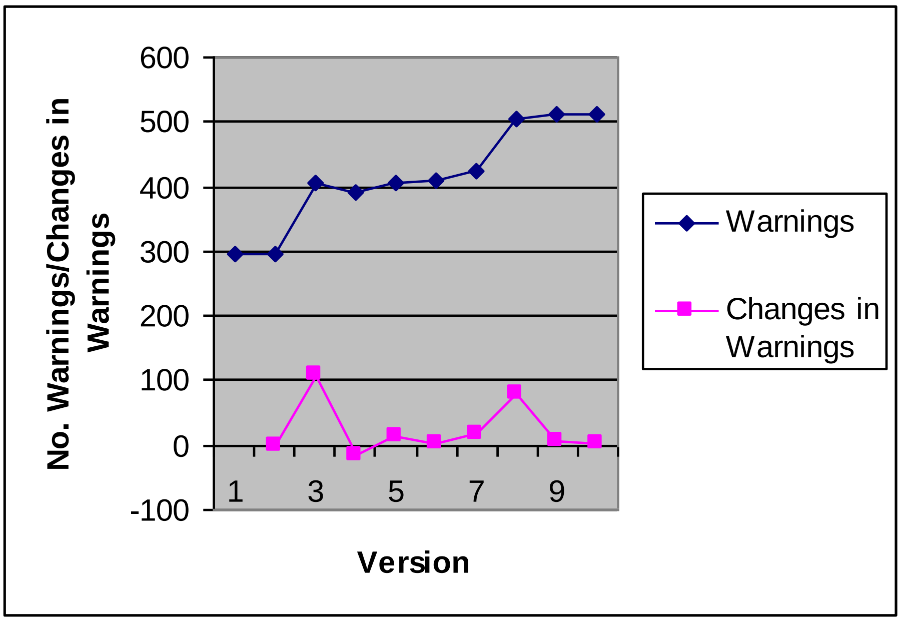

Figure 6.3 Total warnings for pBeans

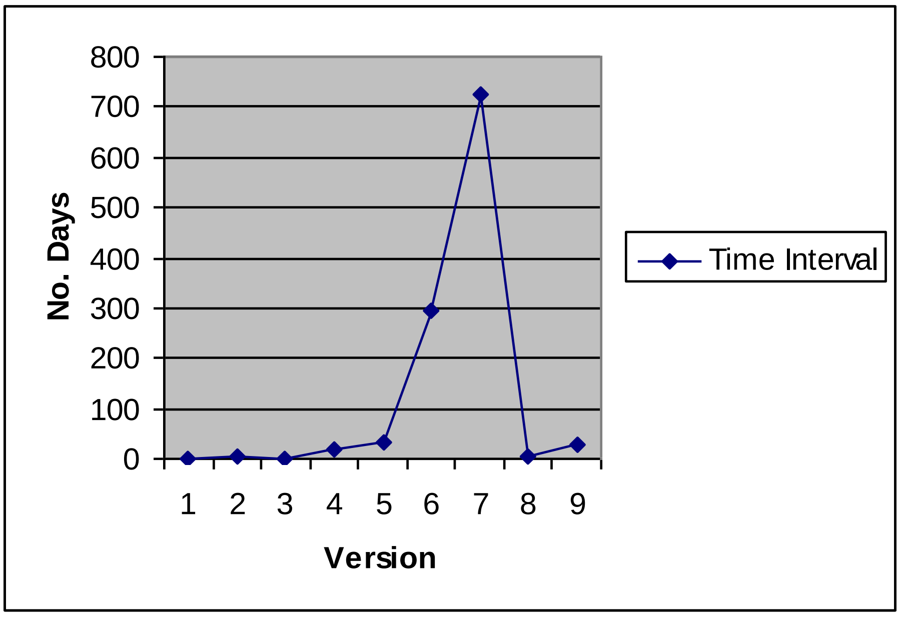

Figure 6.4 Time interval between versions

It is also interesting to note from Figures 6.3 and 6.4 that addition of coupling within a very short space of time, for example in versions 1-3, seems to cause a higher proportion of warnings than when a longer time is spent between versions. Versions 1-3 of the pBeans system were released within a time period of just *seven* days and the same period saw the highest rise in warnings as a result. The overall theme that runs through changes to the pBeans system is that packages may offer a level of encapsulation from access by classes in other packages and, when added anew, do not seem to be the source of significant rises in code warnings.

6.4.3 The SmallSQL System

Table 6.5 shows the changes in each of the five coupling metrics over 9 versions for the SmallSQL system. There is a clear and notable increase in coupling as a result of the addition of a single package between versions 5 and 6. In contrast to the previous two systems, all values of changes in coupling metrics are positive in value.

Table 6.5 Changes in coupling metrics for the SmallSQL system

SmallSQL	Interval	Packages	RFC	MPC	PACK	FOUT	FIN
V1-V2	7	0	19	31	3	28	166
V2-V3	24	0	34	14	1	33	10
V3-V4	34	0	17	17	2	9	7
V4-V5	41	0	48	61	1	6	10
V5-V6	39	1	1055	1698	44	383	169
V6-V7	112	1	109	130	8	23	40
V7-V8	159	0	150	236	5	45	54
V8-V9	70	0	65	110	2	16	20

Table 6.6 presents the warnings categorised in six sets for the SmallSQL system. Figure 6.5 shows the total warnings and changes in number of warnings for that system. In common with the result for the pBeans system, there seems to be only a small effect on the number of warnings from such a large increase in coupling (between versions 5 and 6). The largest rise in warnings comes earlier, between versions 2 and 3, where the time interval between versions was relatively small (24 days). We would have expected a higher rise in warnings following the rise in coupling from version 5 to 6, but this does not seem to be the case. This result supports the claim made for the pBeans system with respect to addition of new packages and the negligible effect that had on generated warnings.

We also note a strong correspondence between the trend for changes in warnings for the pBeans and SmallSQL systems (Figure 6.3 and Figure 6.5). In each case, there is a small peak in warning changes between an early pair of versions and two later versions. The graphs representing the total warnings are also similar and each has a large time interval towards the end of the versions studied (Figure 6.4 and Figure 6.6). Evidence from the pBeans system suggested that addition of new packages may thus have relatively insignificant effects on the number of warnings but that addition of coupling without the addition of packages can create problems. We could thus suggest that, from the combined evidence presented in Figure 6.3 and Figure 6.5, adding coupling in existing packages may have a greater adverse effect on generated warnings than through creation of new functionality. The evidence presented for the previous two systems also supports the claim that addition of new functionality has a lesser effect on potential warnings than modification of existing code.

Table 6.6.6 Warning for Small SQL

SmallSQL	BP	CORR	MCV	MTC	PER	DODGY	Total
V1	47	1	0	0	7	12	38
V2	47	1	0	0	6	12	41
V3	53	1	0	0	6	13	67
V4	53	1	0	0	6	13	66
V5	53	1	0	0	6	13	73
V6	65	1	0	0	5	16	73
V7	65	1	0	0	5	16	73
V8	72	1	0	0	5	16	87
V9	72	1	0	0	5	16	87

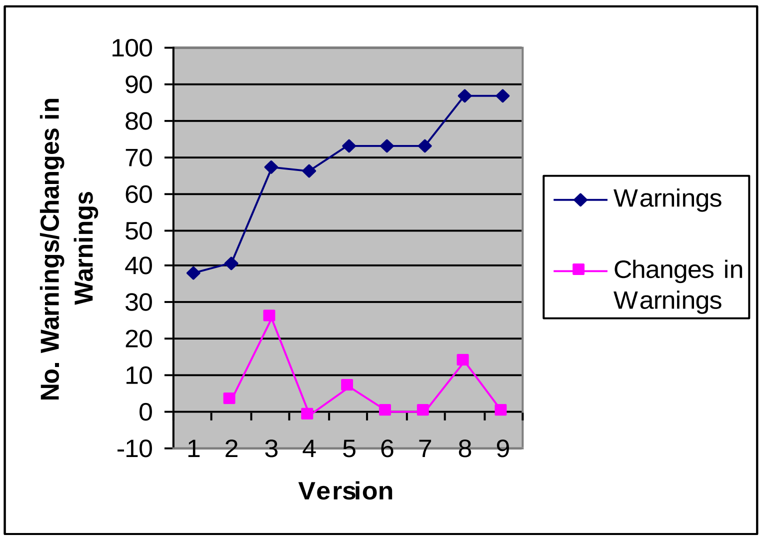

Figure 6.5 Total warnings for SmallSQL

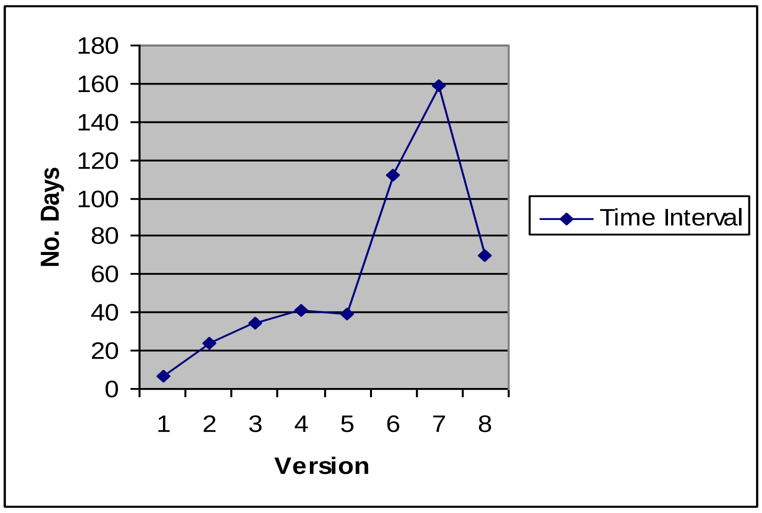

Figure 6.6 Time intervals between versions

6.4.4 The JWNL System

Table 6.7 shows the changes in each of the five coupling metrics over the six versions of the JWNL system. Version 2 to 3 was a simple patch to the system and hence we have omitted coupling values from our analysis in this case (denoted by 'n/a' values).

Table 6.6.7 Changes in coupling metrics for the JMNL system

JWNL	Interval	Packages	RFC	MPC	PACK	FOUT	FIN
V1-V2	0	1	-830	-599	69	-165	-53
V2-V3	n/a	n/a	n/a	n/a	n/a	n/a	n/a
V3-V4	386	9	1286	957	-87	352	232
V4-V5	20	0	0	0	0	0	0
V5-V6	284	-6	-129	-125	173	-150	-73

It is noteworthy that even though only a single package was added to the system from version 1 to 2, a significant fall in coupling was observed for this system.

This was also accompanied by a fall in generated warnings from 166 to 63 over the same versions. It is also interesting to note that each of the coupling metrics saw a decrease in value except for PACK, which suggests that any re-engineering effort that saw RFC, MPC, FIN and FOUT coupling reduced was 'devolved' to other imported packages. The strange feature of the JWNL system is that after version 2 an opposite effect occurred. This process is reversed once again between versions 5 and 6, suggesting that there is distinct contradictory choice being made each time; either use coupling within a package or import that coupling through other packages.

Table 6.8 shows the warning of each of the six categories for the JWNL system. However, Figure 6.7 shows the number of warnings and the changes in warnings for the JWNL system; warnings and change in warnings seem to be rising in parallel after version 4.

Table 6.8 Warning for JWNL

JWNL	BP	CORR	MCV	MTC	PER	DODGY	Total
V1	55	9	52	9	20	21	166
V2	17	2	28	4	8	4	63
V3	18	4	12	2	7	4	47
V4	21	1	13	1	4	4	44
V5	21	1	13	1	4	4	44
V6	33	4	21	3	6	15	82

We could tentatively suggest from the observed data that choice of alternative forms of coupling represent a trade-off between those different types. For example, it has been shown that coupling in the form of C++ friends are correlated with faults (Briand et al., 1997) and such practice should be discouraged as a violation of encapsulation principles; on the other hand, inheritance-based coupling is encouraged when appropriate as good practice. When used to access methods of a class, friends are an alternative to the use of inheritance. Consequently, when choosing to use friends, a developer automatically precludes the choice of inheritance to carry out the same task/s. In theory, reuse coupling through the importing of packages is an essential and unavoidable part of any system (it obviates the need for introducing internal coupling). It is the extent of that importation that seems to make a difference.

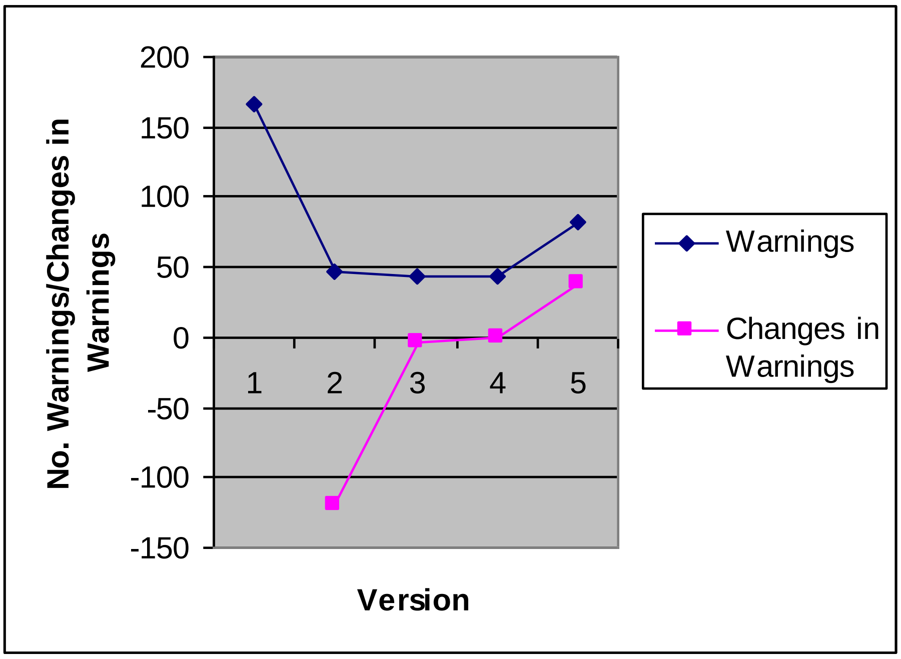

Figure 6.7 Total warnings for JWNL

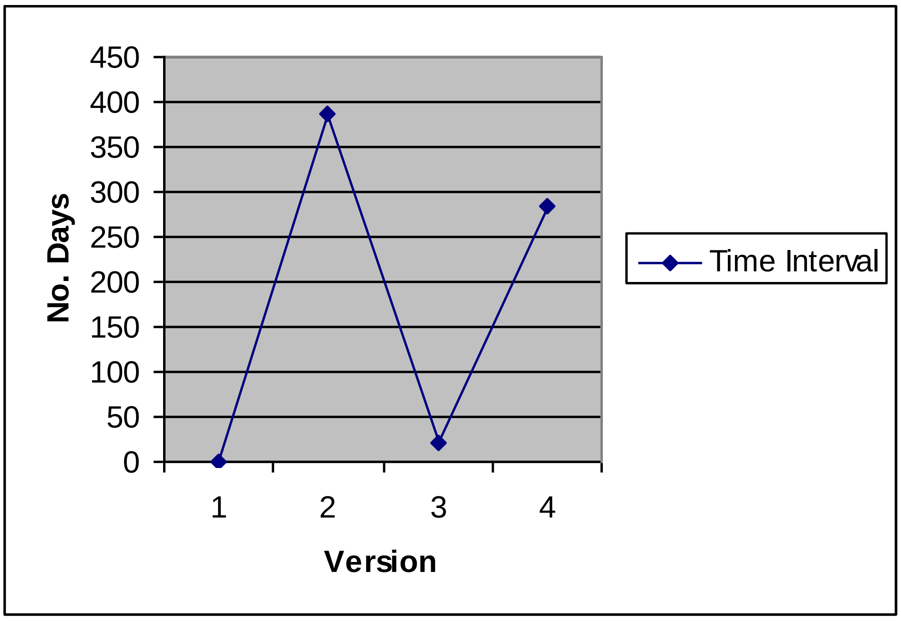

Figure 6.8 Intervals between versions

From the data presented for the JWNL system, it would appear that the practice of inter-changing one type of coupling for another between versions could be the source of potential subsequent problems; more specifically, removing packages may have an adverse effect in terms of warnings.

6.4.5 The DjVu System

Table 6.5 shows the changes in each of the five coupling metrics over the eight versions of the DjVu system. In common with all the previous systems (apart from Jasmin), the addition of packages causes significant increases in coupling metric values. Of the five systems studied, DjVu appears to be the most stable in terms of both warnings and changes in number of warnings (Table 6.10, Figure 6.9). One feature of the data for the DjVu system stands out from all the other systems and might explain this characteristic. Over the course of its versions, only two classes were added to the system even though two new packages were introduced (versions 4 to 6).

Table 6.6.9 Changes in coupling metrics for the DjVu system

DjVu	Interval	Packages	RFC	MPC	PACK	FOUT	FIN
V1-V2	160	0	27	28	1	9	5
V2-V3	41	0	198	110	7	30	109
V3-V4	1	0	0	3	0	2	3
V4-V5	7	1	132	140	10	42	25
V5-V6	18	1	41	72	0	35	22
V6-V7	25	0	-38	-25	11	7	55
V7-V8	40	0	1	0	0	-3	-10

Inspection of the data also revealed that correspondingly few methods were added to existing classes over the course of the versions studied. This very slight increase in classes contrasts heavily with the other four systems (where large numbers of classes and methods were added consistently across versions). In other words, effort in this system seems to have been applied to re-engineer existing classes rather than introduction of new ones. A further notable feature of the DjVu system is the relatively long time interval between releases of earlier versions of the system. The pattern in Figure 6.10 is shared only with the Jasmin system and is characterised by a long time interval between version 1 and 2 (160 days).

Table 6.6.10 Warning for DjVu

DiVu	BP	CORR	MCV	MTC	PER	DODGY	Total
V1	35	9	62	15	58	12	191
V2	35	10	62	15	58	12	192
V3	37	9	56	12	50	9	173
V4	37	9	57	12	50	9	174
V5	36	8	58	12	50	9	173
V6	37	9	60	15	51	9	181
V7	34	7	64	14	51	13	183
V8	34	7	64	14	51	13	183

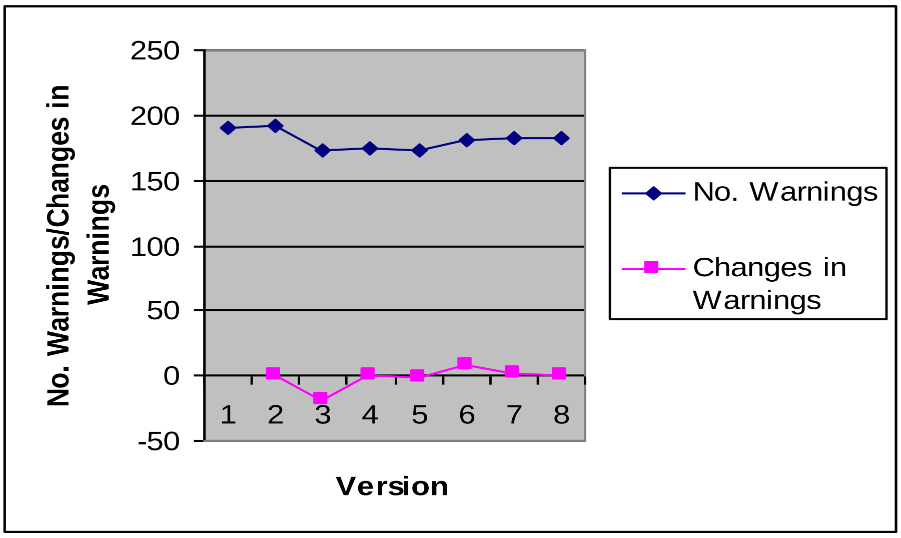

Figure 6.9 Warnings for DjVu

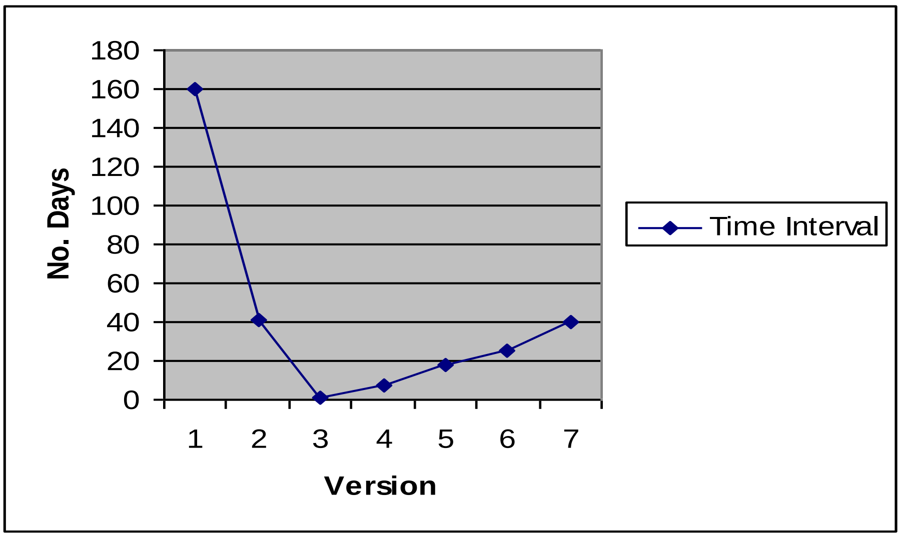

Figure 6.10 Intervals between versions

We could suggest that relative stability may be linked with two characteristics: careful re-engineering of existing code (in terms of time spent) and minimisation of added functionality to existing code. It is also interesting that the Jasmin system is the only other system with a V-shaped time interval curve over the course of its life so far – the other systems (Figures 6.4, 6.6 and 6.8) all approximate an inverted V-shaped curve. We could suggest that spending relatively large amounts of time and care over initial versions of a system and then again applying the same attention later on in a system's lifetime (characterised by the V-curve) may contribute to the stability of a system.

6.5 Discussion

There are a number of implications of the results described in this chapter and a number of threats to its validity. In this chapter, we have tried to relate the analysis to time between versions wherever possible. One feature that every system seems to exhibit is an extreme burst of increased coupling at some point and, usually, within a relatively short time period. Figure 6.11 and 6.12 illustrate the extent of these bursts of activity and, specifically, the significance of coupling peaks for the RFC and MPC metrics for all five systems (when all versions are arranged sequentially). The order from left to right in Figure 6.11 and 6.12 thus represents the RFC and MPC values in the same order of the five systems introduced in Sections 6.4.1 to 6.4.5.

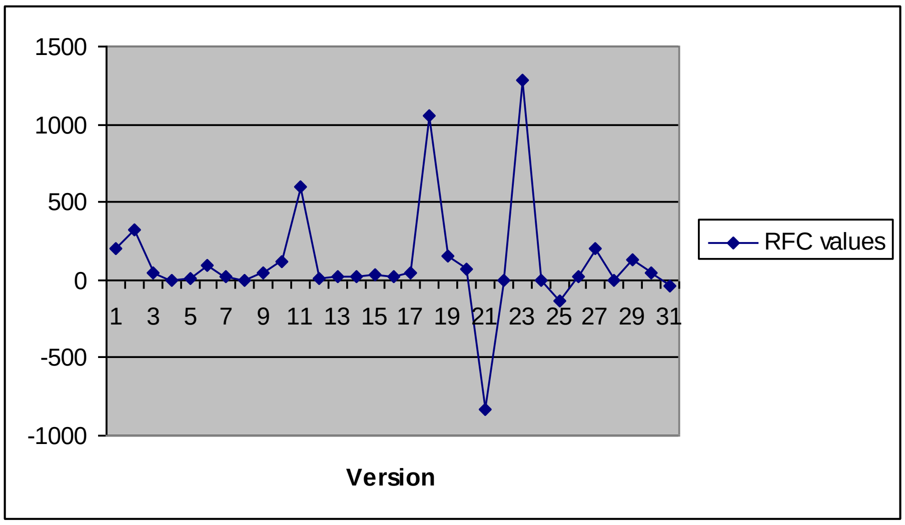

Figure 6.11 Trends in RFC

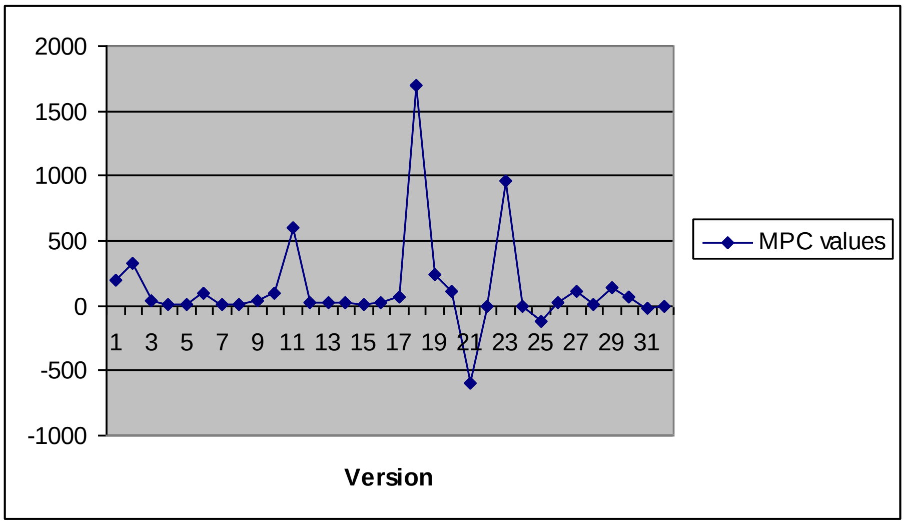

Figure 6.12 Trends in MPC

Most noticeable from Figures 6.11 and 6.12 are the values for the JWNL system which show wide fluctuations in both a positive and negative direction. As shown, at the far right-hand side of each figure, the stability of the DjVu is represented by relatively small peaks. While the claims that we have made in the previous section may be based on single observations and intuition, it is clear that first, total coupling always rises in a system and second, bursts of coupling activity seem to be a characteristic of every system studied.

Since physical time plays such an important role in our analysis, it is worth investigating the possibility that Self-Organized Criticality (SOC), or in other words, whether an 80/20 rule (i.e. 80% of coupling is added in 20% of the time) applies to the addition of coupling over the versions of the systems studied (Wu et al., 2007). If we now consider just the RFC values for each system, then for the Jasmin system, 20% of the total time interval is approximately 129 days. In the 108 days between V2 and V4, only 64% of coupling was added (other short time intervals only add marginally to overall coupling). For the pBeans system, 20% of the time interval is 223 days. 64% of the coupling for this system was added between V6 and V8 where the time interval was 1024 days, suggesting, as for Jasmin, the absence of any 80/20 rule. For the SmallSQL system, 20% of the time interval is approximately 97 days. In the 80 days between V4 and V6, 74% of coupling was added, suggesting a profile more akin to 80/20 (although still falling just short of the threshold if we consider the extra 17 days). For neither the JWNL nor DjVu system is there any evident 80/20 relationship. No significant 80/20 rule is obvious for any of the five systems. The fact that we are only considering added coupling, and not other added data or behaviour may contribute to this lack of empirical support. However, it does further emphasise the enigmatic characteristics of system coupling. In the next chapter, the 80/20 relationship will be investigated in more detail to see whether 80% of total coupling is contained in the top 20% of classes for multiple versions of open-source software and, if so, whether that relationship is exacerbated over time.

A number of threats to the validity of the study also need to be considered. First, we have only used five, medium-sized open-source systems as part of our study. While that provides a cross-sectional view of systems, we accept that this limited number and system size threatens the generalisability of the results. Another threat to the validity of the study is that we have used warnings as a basis of our analysis and not actual faults or complementary techniques (Zheng et al., 2006). However, we feel that it is better to be 'fore-warned' and therefore 'fore-armed' of potential problems and to analyse that data, than to analyse data in a post-fault sense.

The second threat is that many of the warnings issued by FindBugs suggest refactorings that can be applied to remedy the potential problems in the code and so we see our analysis as a contributor the refactoring process. For example, one of the warnings on performance suggests refactoring a class into a named static inner class, if it does not use existing objects appropriately. We note that the majority of warnings for the five systems studied fell into the performance category.

A third threat to the validity of the study is that we have assumed developer activity to be constant throughout the time period studied. This means that on each day there is the same probability of activity on the project. In reality, this might not be the case; a detailed study of developer activity in each system will feature in future work.

The fourth threat is that we have assumed that one package is identical to any other package. In reality, there may be a combination of both user-defined and library-based packages being imported into a system. This analysis will be the subject of future work.

The final threat considered that we have only collected five coupling metrics from a wide range of available coupling metrics in the literature. We defend this choice on the basis that these five provide a set of metrics that allow different levels of code and design abstraction to be analysed and compared, which is a key objective of the study presented.

6.6 Summary

In this chapter, we have investigated trends in coupling in five Java systems. Five coupling metrics were collected from five Java open-source systems using the JHawk tool and warnings for each version collected using the FindBugs tool.

Investigation of the five systems revealed a common trend of bursts of additional coupling and the emergence of a number of themes. First, and surprisingly, the addition of coupling may have beneficial effects on a system. Second, and more intuitively, it seems that the addition of coupling in new functionality through addition of packages could result in fewer warnings than adding functionality to existing code. Finally, there appears to be a trade-off between coupling types, in particular, that between couplings through imported packages and the introduction of internal-to-the-package coupling.

In the next chapter, the notion of an 80/20 relationship discussed in this chapter will be presented in more detail. The coupling metrics will be tested to see if they obey the 80/20 rules in the class basis. The top 20% of classes will be explored to see if they contain the 80% of the coupling. Moreover, in the next chapter we will investigate the relationship between the FIN and FOUT metrics to see whether they increase in corresponding amount and consistently over time, and to investigate the characteristics of classes exhibiting the highest values of these two metrics.

CHAPTER 7. EVOLUTIONARY STUDY OF FIN AND FOUT

7.1 Introduction

One observation made from the studies in Chapters 4, 5 and 6 was that the bulk of changes and coupling activity (identified by the metrics collected) tended to focus around a small number of classes, while on the other hand the vast majority of classes remained untouched throughout the same versions studied.

Pareto’s Law or an ‘80/20’ rule as it is sometimes known is a naturally occurring phenomenon. For example, we could claim that 80% of floods comprise just 20% of the total destructive damage around the world. Unfortunately, and sadly, the other side of the coin is that 20% of floods (the most severe and destructive ones) account for 80% of the total damage. In the context of the Thesis, we might suggest that 80% of class activity in a system occurs in just 20% of classes. In the previous chapter, coupling in five Java systems using five coupling metrics, version release times and code warnings was empirically explored. The results that were reported in that chapter revealed a common trend of bursts of additional coupling and suggested that coupling is a multi-faceted, multi-dimensional and more complex feature of a system than may have been appreciated in the past.

Moreover, in the previous chapter, there was a brief investigation to see whether an 80/20 rule applied to the addition of coupling over the versions of the systems studied. In this chapter, this investigation is studied in more detail. We explore whether an 80/20 rule exists in Java from six coupling metrics over multiple versions of open-source software and, if so, whether that relationship is exacerbated over time. The automated tool JHawk was used to extract the six different coupling metrics from four Open-Source Systems. Afterwards, the classes were ranked on each of these 6 coupling metrics and then the top 20% of classes were analysed to see whether 80% of total coupling was contained therein. Only one metric appeared consistently to have an 80/20 relationship and that was the FIN metric. Evidence suggests that FIN and FOUT have a complementary relationship. We found many of the other metrics had few, if any, such relationships. We also found no evidence to support the view that over time, the 80/20 is exacerbated. The relationship between FIN and FOUT coupling metrics suggested another investigation, so we explore this relationship over multiple versions of open-source software. More specifically, we explore the relationship between the two metrics to determine patterns of growth in each over the course of time. Two questions were posed for each system. First, what are the characteristics of classes exhibiting the highest FIN values? Second, do FIN and FOUT increase in corresponding and consistent amounts over time? Results show a wide range of traits in the classes to explain both high and low levels of FIN and FOUT. We also found evidence of certain 'key' classes (with both high FIN and FOUT) and 'client' and 'server'-type classes with just high FOUT and FIN, respectively. We provide an explanation of the composition and existence of such classes as well as for disproportionate increases in each of the two metrics over time.

The remainder of this Chapter is structured as follows. In the next section, we present the motivation for the study and related issues. In Section 7.3, we provide details of the systems studied, the data collected and the research questions. Section 7.4 includes an analysis of each system individually in order to assess the research questions. Finally, we conclude and summarise the study in Section 7.5. We note that the research in this chapter was first published by Mubarak et al. (2009) and also in Counsell et al. (2010).

7.2 Study Motivation and Related Issues

Many social and naturally occurring phenomena are distributed according to an 80/20 rule (sometimes known as a Power Law). In other words, 'small' occurrences of an artefact or phenomenon are extremely common, whereas 'large' instances are relatively rare. Wheeldon and Counsell (2003) illustrated that a Power Law distribution existed in OO class relationships, particularly those related to coupling (via inheritance and aggregation). In this chapter, we attempt to support or refute that earlier work by focusing on 6 separate, yet different coupling metrics to explore whether evolutionary coupling obeyed an 80/20 rule. In theory, we would expect coupling to increase through consistent application of maintenance as a system evolves and, hence, for any 80/20 rule to become exacerbated. The study presented also attempts to shed light on which coupling *types* tend to exhibit specific trends (in this case an 80/20 rule). There are also parallels with the use and credibility of 'key' classes, i.e. certain classes in any system that comprise a large number of methods (and, by implication, a large amount of coupling).

In practice, a class that is highly coupled to many other classes is an ideal candidate for re-engineering or removal from the system to mitigate both current and potential future problems. A problem that immediately arises, however, for the developer when considering re-engineering of classes with high coupling is: 'Do those classes have prohibitively large dependencies?' If so, then are those coupling dependencies 'incoming' or 'outgoing' dependencies? In theory, it is more difficult to modify a target class with high incoming and low outgoing coupling, since the former requires detailed and careful scrutiny of each of the many 'incoming' dependent classes and the possible side-effects of change. Chapter 5 showed that the FIN and FOUT metrics tended to be relatively small for classes removed from a system. In other words, classes with either high FIN and/or FOUT may be difficult to move or remove from a system. This question has inspired further examination of trends in the two metrics presented.

Chapter 6 has shown that there is a trade-off between coupling types – in particular, that between coupling through imported packages and the introduction of 'internal-to-the-package' coupling. In this chapter, we explore the potential characteristics and trade-offs between FIN and FOUT metrics over time. We would always expect potentially problematic classes to be re-engineered by developers through techniques such as refactoring (Fowler, 1999); however, the practical realities of limited time and resources at their disposal means that only when classes exhibit particularly bad 'smells' (e.g. excessive coupling) (Fowler, 1999) are they dealt with.

In this chapter, we address the issue of potential re-engineering and view coupling as a key contributor to the decision on whether and when to re-engineer (classes) or not over the lifetime of a system. Chapter 4 showed some evidence to suggest that 'peaks and troughs' occur in software maintenance, suggesting that developer activity comprises a set of high and low activity periods. This suggests that excessive coupling is a continuous problem addressed only by spurious and frenzied re-engineering activity.

7.3 Systems and Metrics

7.3.1 Systems under Study

Five systems were used as a basis of our study. These systems were used in the study conducted in Chapter 5 and Chapter 6, and they were presented in Section 3.4.1. These systems are Jasmin, DjVu, pBeans, SmallSQL and Asterisk.

7.3.2 Data Collected

For each of the systems, we collected six independent, coupling metrics using JHawk (2008) (as described in Section 3.4.2). These metrics are as follows:

1. Response for a Class (RFC): The RFC is defined as the set of methods that can potentially be executed in response to a message received by an object of that class.

2. Number of EXTernal methods called (EXT): The more external methods that a class calls, the more tightly bound that class is to other classes

3. Message Passing Coupling (MPC): The number of messages passed among objects of a class.

4. PACK. The number of imported PACKages.

5. Fan-in (FIN). The FIN of a function is the number of unique functions that call the function.

6. Fan-out (FOUT). The FOUT is the number of unique functions that a function calls.

7.3.3 Research Questions

The study consists of two research questions (RQ1 and RQ2), stated as follows:

- **RQ1.** Does an 80/20 rule exist in Java from six coupling metrics over multiple versions of open-source software? If so, is that relationship exacerbated over time? We try to see if the six coupling metrics obey the 80/20 rule by discovering whether the top 20% of the classes contain at least 80% of the coupling metrics or not.
- **RQ2.** Is there a significant correlation between FIN and FOUT? If so, does this relationship worsen over time? If the correlation is negative, then this suggests that, over time, an inverse relationship exists between the two metrics. In other words, as FIN increases, there is a decrease in the value of FOUT and *vice versa*. On the other hand, a positive correlation between the two metrics would imply that both FIN and FOUT increase as a system evolves. As a developer, we would want to choose classes/packages for re-engineering in the former category and preferably when FOUT is increasing and FIN decreasing.

We note that in the following, we use three correlation coefficients. Spearman's and Kendall's coefficients are non-parametric in nature and assume a non-normal distribution in the data (appropriate for most software engineering data). For completeness, however, we have also included Pearson's correlation values – a parametric value which assumes a normal distribution of the data. The FIN and FOUT values for all selected classes and for each version were used as a basis of the correlation analysis.

7.4 Data Analysis

In order to assess our research questions, we collected six coupling metrics for the five systems. For the first research question, we calculate the percentage of the coupling metrics per package for each class over the versions of four OSSs (Jasmin, DjVu, pBeans and SmallSQL). However, for the second research question we calculate the correlations between the FIN and FOUT over the versions of all the five systems. We also compare the differences between the FIN and FOUT of the classes and the mean of these metrics across the whole package. We assess each question on the systems separately.

7.4.1 Research Question 1 (RQ1)

In the following analysis and for succinctness, we include in the tables for each system *only* the rows where at least one 80/20 rule was found to apply; equally, only the columns (i.e. metrics) where at least one 80/20 rule was found to apply are listed. If a value is omitted from a table, then no 80/20 rule is applied in that case. (We note that for each system, the top 20% of classes will be *exactly* 20% of the total number of classes stated earlier in the description of the systems.) An 80/20 rule applies *if* at least 80% of the coupling is incorporated in that top 20%.

7.4.1.1 The Jasmin System

Table 7.1 shows the percentage in each of the coupling metrics over the five versions (V1-V5) of the Jasmin system for 20% of the classes on a package basis (the 2 packages in this case are jas and jasmin).

The most striking feature of the values in Table 7.1 is the absence of four of the six metrics extracted by the tool and subsequently analysed. No entries for RFC, EXT, PACK or FOUT greater than or equal to the threshold 80% were found.

In V1 of the jas package, the top 20% of the classes comprised over 90% of all FIN coupling. A comparison of V1 and V5 shows that the 80/20 rule became weaker by V5 (i.e. at 84.78%). For the jasmin package, there is only a marginal increase in the 80/20 rule (from 90.71% to 93.33%). It is also worth noting that the FOUT metric had many values between 70% and 75% over the course of these versions and consequently are not shown in the table. Equally, the values of the other four metrics tended to be in the range 45%-70%, considerably lower than the FIN values.

Table 7.1 80/20 metrics for the Jasmin system

Package	Version	MPC	FIN
Jas	V1		90.15
	V2		89.35
	V3		83.41
	V4		84.78
	V5		84.78
Jasmin	V1		90.71
	V2		92.9
	V3	81.39	94.19
	V4		92.9
	V5		93.33

7.4.1.2 The SmallSQL system

Table 7.2 shows the same data for the SmallSQL system. In common with the Jasmin system, the FIN metric satisfied the 80/20 rule across all versions studied. However, the rule is only marginally strengthened between V1 and V9 (88.54% to 92%) - there is no support for the view that evolution of the 80/20 rule is strengthened.

Table 7.2 80/20 metrics for the SmallSQL system

Package	Version	FIN
Database	V1	88.54
	V2	90.33
	V3	91.18
	V4	91.15
	V5	91.23
	V6	92.44
	V7	92.19
	V8	92.11
	V9	92.00

7.4.1.3 The DjVu System

Table 7.3 shows the data for the DjVu system. Again, FIN appears prominently in the table and so too does the FOUT metric. However, the PACK metric features in V8 of the DjVu package. One plausible explanation for the dominance of FIN and FOUT and an implication of that dominance is that it may render the use of other coupling forms unnecessary.

It is interesting that none of the values in Table 7.3 overlap. Inspection of the raw data revealed that generally, when FIN was high, FOUT was low (and vice-versa)

Table 7.3 80/20 metrics for the DjVu system

Package	Version	PACK	FOUT	FIN
DjVu	V5		80.23	
	V6		80.23	
	V7		80.46	
	V8	80.68		
Toolbar	V1			83.81
	V2			81.90

7.4.1.4 The pBeans System

Table 7.4 shows the trends for the pBeans system. The FIN metric does not feature in the 80/20 rule in the pBeans package. It does, however, feature in the first eight versions of the data package. This suggests that the FIN and other forms of coupling may have a complementary relationship. When there is a high proportion of 80/20 FIN relationships, there is a low number of other 80/20 forms of coupling. Tables 7.1 and 7.2 support this theory and in Table 7.3 the six 80/20 relationships are non-overlapping (further supporting this theory). One explanation for this phenomenon, in a practical sense, is that if there are a high number of classes with large FIN values, then, by the *law of averages,* there will be fewer classes with large numbers of FOUTs since the coupling that these latter classes need is satisfied by the classes with large FIN (and which they use). In other words, FIN and FOUT follow a 'hub' principle, which minimises outgoings by maximising incomings. This would also support the use of 'key' classes (i.e. classes that contain a large amount of functionality that many other classes use) as a means of minimising coupling. In other words, and counter-intuitively, large classes (or classes with large amounts of coupling) can have a beneficial effect (if we assume minimising FOUT is an aspiration of developers).

Table 7.4 80/20 metrics for the pBeans system

Package	Version	MPC	EXT	PACK	FOUT	FIN
pBeans	V1	90.98	85.73	85.00	88.32	
	V2	90.79	85.67	85.00	88.32	
	V3	91.89	86.56	81.05	87.29	
	V4	91.89	86.56	81.05	87.29	
	V5	92.23	87.10	81.05	87.29	
	V6	92.23	87.10	81.05	87.29	
	V7	94.02	89.23	81.82	90.60	
	V8	89.50	84.67	83.90	85.69	
	V9	89.50	84.67	83.90	85.69	
	V10	89.54	84.74	83.90	85.69	
Data	V1					82.86
	V2					87.50
	V3					92.89
	V4					92.89
	V5					92.89
	V6					94.78
	V7					99.53
	V8	89.64	85.08		83.76	81.14
	V9	88.86	84.38		80.22	
	V10	88.26	83.85			

Figure 7.1 shows the pBeans metrics over the course of *all* 20 versions from Table 7.4. The values follow the same trends for much of all 20 versions (except for RFC and FIN). Low values in the figure reflect a more even spread of coupling for that metric.

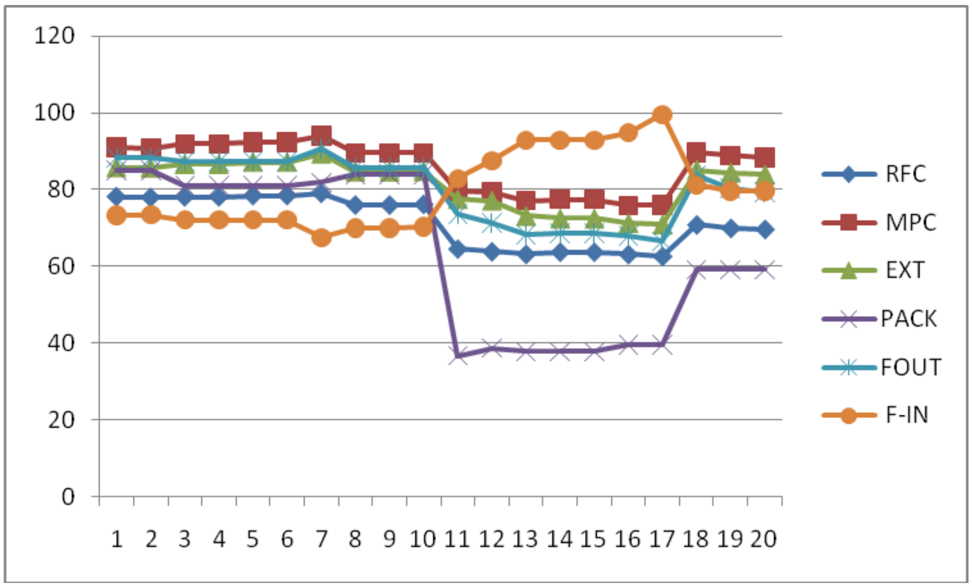

Figure 7.1 Metric values for pBeans

Figure 7.2 shows the same values for the SmallSQL system. Coupling types appear to have the same pattern in both figures; inspection of the raw data for the other two systems revealed a similar trend. In other words, coupling remains relatively static for all systems as they evolve; an 80/20 rule is not exacerbated as a system evolves.

We can conclude for the first research question that certain metrics had a greater propensity for that rule than others, namely FIN and, to a limited extent, FOUT. High use of these two features seemed to exclude the use of other types of coupling. Moreover, an 80/20 rule did not seem to worsen as a system evolved. Finally, we suggested that dominance of FIN (particularly) might act as a ‘hub’ for ‘key’ classes and with which many other system classes communicate.

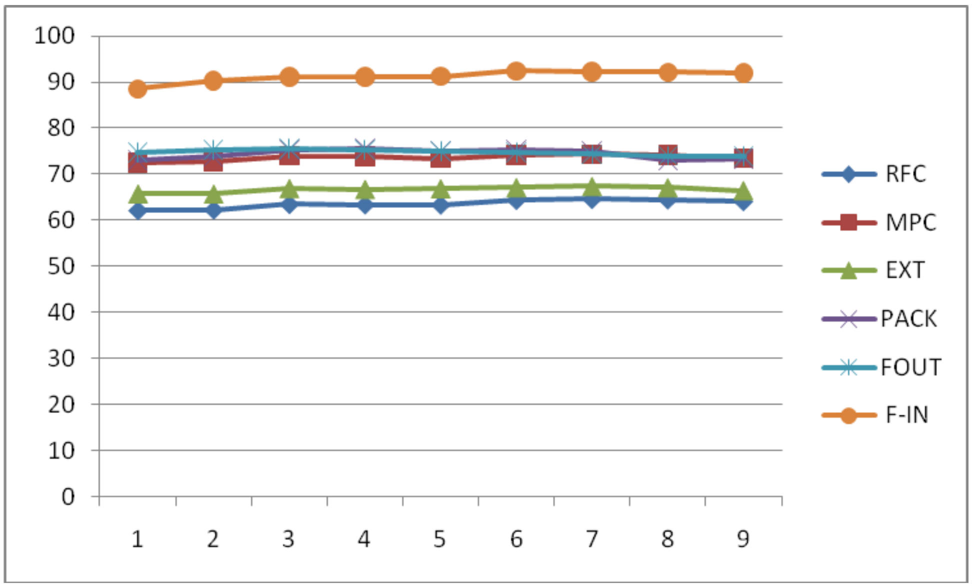

Figure 7.2 Metric values for SmallSQL

7.4.2 Research Question 2 (RQ2)

In the following analysis, we consider only the largest packages from each system. A package was considered as ‘large’ if it contained more than ten classes (for statistical validity purposes, we wanted to ensure that the number in each package was relatively high and ten seemed a reasonable threshold). We ranked the classes in each of these packages according to their descending FIN values and then took the set of classes from each package that contained 80% of the FIN total. We chose the classes comprising 80% of FIN for a single reason. The previous research question has shown that an 80/20 rule applies to coupling in Java classes. In other words, 80% of FIN occurs in just 20% of classes. To be true to the spirit of that earlier research question, we adopted the same strategy for selection of classes. Moreover, we wanted to focus on classes with a high FIN and choice of classes comprising 80% of the FIN, when ordered in descending FIN captures classes with the highest FIN.

Additionally, choosing classes comprising 80% of FIN would also allow us to compare (i.e. correlate) the FIN of those classes with the FOUT of the same set of classes to establish overall relationships between the two metrics and to uncover biases in class make-up and disparity between the two metrics. In particular, we would like to explore the presence of 'key' classes characterised by a high FIN and high FOUT value, as well as to distinguish 'server' classes that have a high FIN (i.e. they are used by many classes) but a low FOUT (i.e. they correspondingly do not use many other classes themselves). The profile of these types of classes from an evolutionary perspective is also an interesting research topic and one that we explore.

The mean of the FIN and FOUT *across the whole package* was also calculated to allow a comparison of the differences between the selected classes and the summary values of FIN and FOUT *for all classes* on an evolutionary package basis.

7.4.2.1 The Jasmin System

We first consider the set of classes comprising the 80% of FIN. Table 7.5 shows the correlation between the FIN and FOUT over the five versions (V1-V5) for the Jasmin system on a package basis (the 2 packages chosen using the aforementioned selection criteria in this case were Jas and Jasmin). Extracting classes containing 80% of FIN from the Jas package gave a sample of 50 classes for that package and 10 classes for the Jasmin package.

The most striking feature of the values in Table 7.5 is the significant positive correlation between the two metrics for Jas package (Kendall's and Spearman's), while the correlation values are strongly and significantly negative for the Jasmin package. There is a simple, yet interesting explanation for each set of correlation values. For the Jas package while the values of FIN are large, the values of FOUT are correspondingly large (see Figure 7.3). Many of these classes are therefore those *used by* many other classes, but also *themselves* use high numbers of other classes. We could thus view this type of class as both a coupling 'source' and 'sink' classes since they use equal measures of both FIN and FOUT. The dependence of many classes on these types of class alone may make them problematic from a re-engineering perspective. Indeed, Figure 7.3 shows very few classes where both the FIN is low and FOUT high which would be one possible and sensible criterion for re-engineering.

For the Jasmin package on the other hand, the FIN and FOUT metrics are in complete contrast (see Figure 7.4). Classes with high FIN values in this package tend to have low FOUT values and vice versa. The classes in this latter category would be far preferable for re-engineering – since high values for FOUT alone pose less of a problem from a maintenance perspective - the dependencies are outgoing rather than incoming.

Table 7.5 Correlations FIN vs. FOUT (Jasmin)

Package	No. of Classes	Pearson's	Kendall's	Spearman's
Jas	50	0.024	0.287**	0.394**
Jasmin	10	-0.973**	-0.619*	-0.788**

*Correlation is significant at the 0.05 level (1-tailed).
**Correlation is significant at the 0.01 level (1-tailed).

From a correlation perspective, both packages present opportunities for re-engineering, but the negative correlations for the Jasmin package provide the best opportunity in this sense and the Jas only limited opportunities. In other words, analysis of coupling through extraction of FIN and FOUT has provided an insight into which classes might be targeted for re-engineering. This would not be the case had we just collected coupling on a far coarser scale using for example, the CBO (Chidamber and Kemerer, 1994). The CBO makes no distinction between input coupling and output coupling.

We next consider the set of *all* classes in each of the two packages. The summary data for FIN and FOUT in Table 7.6 shows the mean and median values for *every* class in each of the studied packages over the five versions (V1-V5).

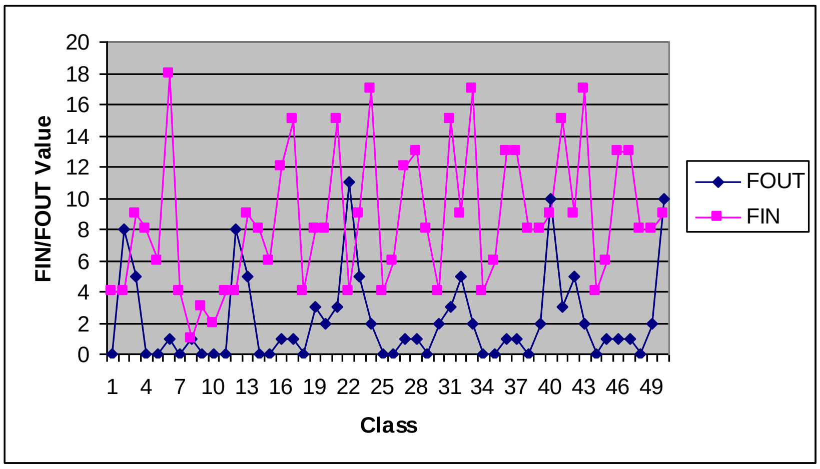

Figure 7.3 FIN/FOUT for the Jas package

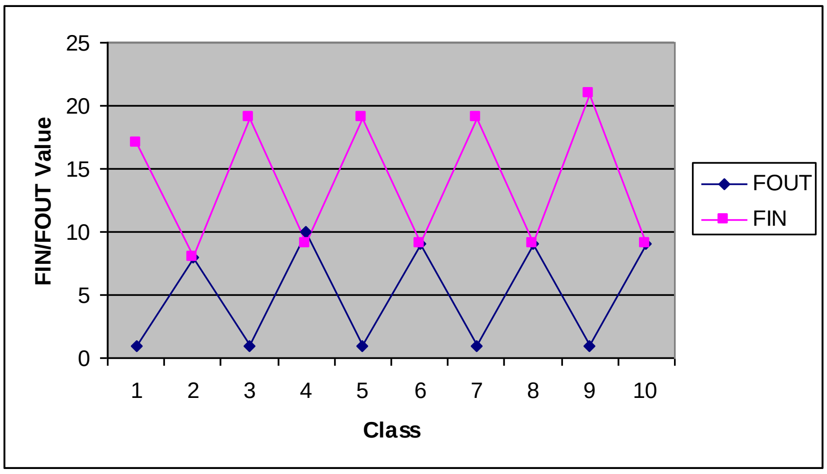

Figure 7.4 FIN/FOUT for the Jasmin package

Table 7.6 shows the values of FIN across all the classes of Jas to be relatively small and so too the values of FOUT. (We note that the values of FIN in each package have been italicised to distinguish them from FOUT values.) There is a clear upward trend in the values of FIN and FOUT in both packages. However, the median values (column 3) do not change significantly throughout and this suggests further that in each package there are certain outliers that subvert the true picture of the FIN and FOUT metrics (i.e. those in Figures 7.3 and 7.4).

The values in Table 7.6 indicate that although FIN and FOUT increase over time, these increases are relatively small. The average values of FOUT in the Jasmin package are significantly higher than that for FIN, again suggesting that classes in this package would be preferable and more amenable to re-engineering than Jas. In answer to the question posed, we see a similarity between the growth in values of FIN and FOUT as they evolve, but not alarmingly so.

Table 7.6 FIN and FOUT per package (Jasmin)

Package	Metric (Ver.)	Mean	Median
Jas	FOUT (V1)	0.67	0
	FOUT (V2)	0.84	0
	FOUT (V3)	1.23	0
	FOUT (V4)	1.35	0
	FOUT (V5)	1.44	0.5
	FIN (V1)	*1.35*	*0*
	FIN (V2)	*1.61*	*0*
	FIN (V3)	*1.86*	*0*
	FIN (V4)	*2.03*	*0*
	FIN (V5)	*2.03*	*0*
Jasmin	FOUT (V1)	6.64	2
	FOUT (V2)	7.17	1.5
	FOUT (V3)	7.23	1
	FOUT (V4)	8.08	1.5
	FOUT (V5)	8.08	1.5
	FIN (V1)	*2.55*	*0*
	FIN (V2)	*2.58*	*0*
	FIN (V3)	*2.38*	*0*
	FIN (V4)	*2.58*	*0*
	FIN (V5)	*2.75*	*0*

7.4.2.2 The SmallSQL System

Table 7.7 shows the same correlation values we showed for Jasmin for the SmallSQL system. Only one package was considered for this system, namely 'Database'. The number of the classes comprising the 80% of FIN is 25 classes from a total 135 classes across the nine versions giving a total sample correlation size of (9*25=225). A positive correlation between the FIN metric and the FOUT is apparent from Table 7.7. However, the correlation is weaker for Kendall's and Spearman's, while there is no significant correlation for Pearson's.

Table 7.7 Correlations FIN vs. FOUT (Small SQL)

Package	Pearson's	Kendall's	Spearman's
Database	0.041	0.130**	0.175**

**Correlation is significant at the 0.01 level (1-tailed).

Figure 7.5 shows the values of FIN and FOUT over the nine versions for SmallSQL system on a package basis. From Figure 7.5 it can be seen, as was seen for the Jasmin system, that there are some classes with exceptionally large values of FIN. One class that is particularly noticeable is the Utils class, which started with a FIN of 245 in V1 and by V9 had a FIN of 416. In contrast, its FOUT started in version 1 with a value of just 12 and rose to only 19 by version nine.

A class such as Utils (as its name suggests) is likely to be used (i.e. 'utilised') and in great demand increasingly as a system evolves and as more classes are added to the system. A Date class for example is found in java.util – a class which his likely to be used by many other classes. Interestingly, the number of methods in this class and its size in terms of LOC did not change significantly. It started with 25 methods and 211 LOC in V1 and in V9 had 34 methods and 257 LOC. In other words, the class itself did not change, but the number of classes *using that class* grew significantly.

While the benefits of such a class are clear, classes such as Utils could conceivably pose a problem for developers. With such a high FIN, it becomes difficult to modify such a class and this might explain why its size in terms of methods and LOC changed only marginally over the nine versions. This type of class could also be seen as a *key* class to the functioning of the system and while stable in some senses, might be exceptionally difficult to re-engineer. On the other hand, the fact that it has not changed significantly over the versions studied may mean that it does not need to be re-engineered – so the potential danger outlined is not germane.

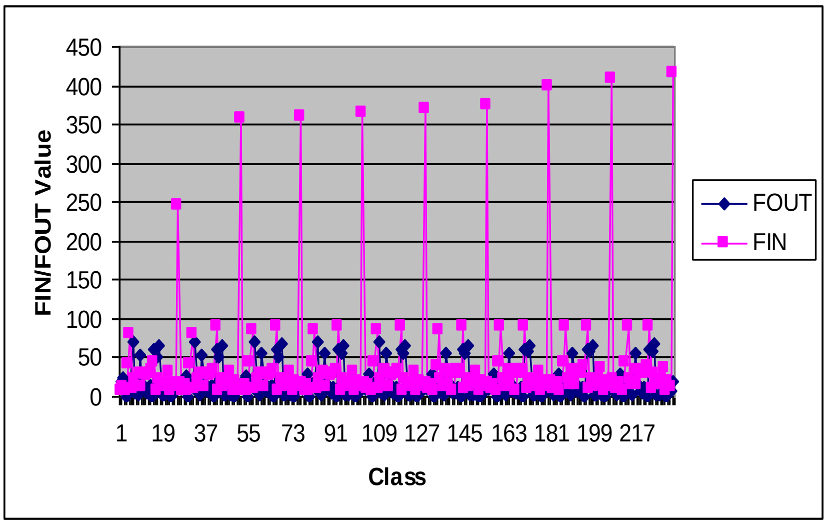

Figure 7.5 FIN/FOUT for the database package

Table 7.8 presents a summary of the FIN and FOUT for all classes in the Database package of SmallSQL and again gives the value of the mean and median values.

The interesting feature of Table 7.8 is the drop in both the FIN and FOUT metrics in the transition from V5 to V6. This was not accompanied by any noticeable reduction in the size of the classes; there was some reduction in coupling however, suggesting that between these versions there may have been some effort devoted to re-engineering (with the consequent drop in coupling). Both FIN and FOUT seemed to mirror each other's movements. This again was interesting since it meant that if FIN changed, then FOUT would be changed as a result and as the system was re-structured. It might also be the case that some active refactoring was undertaken to eliminate inter-class coupling; a natural result of eliminating inter-class coupling is the elimination of total coupling since dependency 'tangling' is simplified overall. In keeping with the Jasmin system, the values of FIN and FOUT remain relatively static. The system does contain, however, a number of classes those are key to the functioning of the system (such as Utils).

Table 7.8 FIN and FOUT per package (SmallSQL)

Metric (Ver.)	Mean	Median
FOUT (V1)	10.32	3
FOUT (V2)	10.53	3
FOUT (V3)	10.31	3
FOUT (V4)	10.38	3
FOUT (V5)	10.42	3
FOUT (V6)	9.96	3
FOUT (V7)	9.98	3
FOUT (V8)	10.13	3
FOUT (V9)	10.19	3
FIN (V1)	*6.92*	*0.5*
FIN (V2)	*8.19*	*0.5*
FIN (V3)	*7.90*	*0*
FIN (V4)	*7.96*	*0*
FIN (V5)	*8.03*	*0*
FIN (V6)	*7.48*	*0*
FIN (V7)	*7.69*	*0*
FIN (V8)	*7.94*	*0*
FIN (V9)	*8.08*	*0*

7.4.2.3 The DjVu System

Table 7.9 shows the data for the DjVu system. The number of classes comprising 80% of the FIN was 8 out of 40 across the 9 versions for the Djvu package (the package shares its name with the system from which it is taken), 2 out of 10 for Anno package and 2 out of 9 for Toolbar package. The number of classes for which we calculated the correlations between FIN and FOUT was 64 for Djvu and 16 apiece for Anno and Toolbar. The first question relates to the nature of the correlations. Negative correlations between FIN and FOUT are evident for the Djvu and Anno packages. There is positive correlation between the metrics for the Toolbar package over the same 8 versions of DjVu system.

Table 7.9 Correlations FIN vs. FOUT (DjVu)

Package	Pearson's	Kendall's	Spearman's
Djvu	-0.088	-0.151	-0.248*
Anno	-0.572*	-0.436*	-0.462*
Toolbar	0.988**	0.914**	0.950**

*Correlation is significant at the 0.05 level (1-tailed).
**Correlation is significant at the 0.01 level (1-tailed).

Figure 7.6 shows the values of FIN and FOUT for the Djvu package. The values of FIN are consistently higher than that of FOUT. The lines in the graph do not overlap at all and are totally disjoint. This feature contrasts with all the graphs shown for the previous two systems.

The class which, over the course of the eight versions was consistently high in its FIN was the GRect class; this class started with a FIN value of 59 and ended with a FIN of 81. It had one of the lowest FOUT values for that package however (value of just 4) throughout the versions studied, compared with a mean of 5.67 for the remaining classes. Again, this might be a class for manipulating GUIs which might be critical to system functionality (i.e. key class). Based on the fact that the DjVu system is graphically-oriented system – we would expect a shapeoriented class to be the subject of significant use by other classes in the system and this might explain its high FIN. It also gives an insight into the way this type of system evolves. A key class does not see any rise in FOUT, but does in terms of its FIN as more classes use the class. GRect thus acts as a server class to other classes.

Figure 7.7 shows the FIN and FOUT values for the Anno package. It is interesting that, there is a striking difference between the FIN and FOUT values towards later versions of the system studied. The class with the FIN of 11 was the Rect class. The values of FOUT remain static between V4 and V5. Interestingly, it seems that in two packages in this system, the same types of class (i.e. rectangle-based) are both prominent classes (GRect and Rect). This supports our view that there are certain classes whose FIN increases because of their popularity and whose FOUT remains relatively static. One conclusion that we could draw from our study is therefore that an increasing FIN is not necessarily a sign of decay as such. Some classes become increasingly used by other classes for the functionality they provide. A class whose FIN increases while its FOUT remains stable is a possible sign of one of these types of class.

Figure 7.8 shows the same data for the Toolbar package. In contrast with any other packages/systems studied, the values of FOUT are significantly *higher* than that of FIN. There is a strong correspondence between the FIN and FOUT for this package and the values of FIN and FOUT mirror each other; each rise and fall correspondingly. This type of class is characterised by the feature that as incoming coupling is added to it, so too is added outgoing coupling and is in contrast to classes such as Rect and GRect just described. This feature may be due to the graphical processing nature of the classes in the DjVu system requiring input from other GUI-based classes and feeding the output to further GUI-based classes – e.g. processing *x* and *y* co-ordinate classes which feature heavily in this system. This would certainly be a plausible explanation. On the other hand, it might be a sign of a relatively balanced system that the FIN and FOUT are correspondingly large.

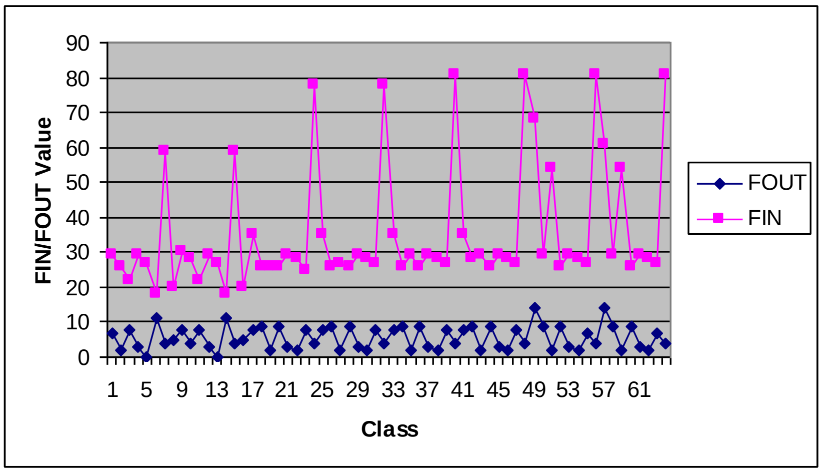

Figure 7.6 FIN/FOUT for the Djvu package

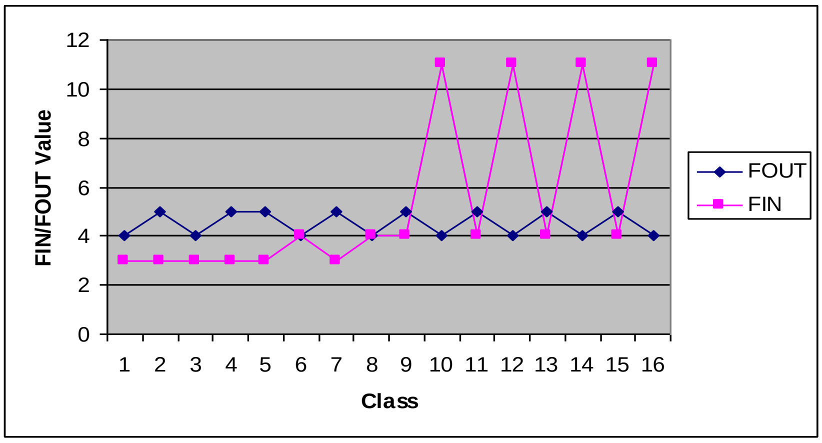

Figure 7.7 FIN/FOUT for the Anno package

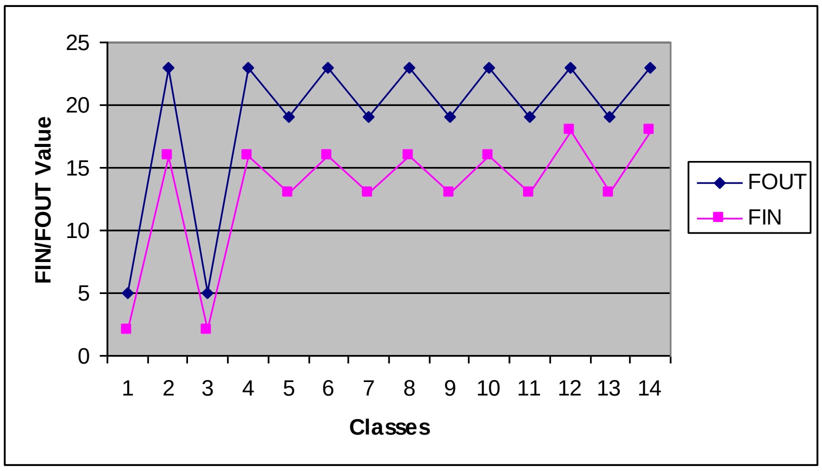

Figure 7.8 FIN/FOUT for the Toolbar package

Table 7.10 shows a summary data for the FIN and FOUT metrics over the different versions for all classes in those packages. (We note that when versions have the same mean and median FOUT or FIN, we list those versions in a single row of the table rather than duplicate a row; see, for example, row 1 of the data in Table 7.10, pertaining to the FOUT for V1 and V2.)

The Toolbar package is the most striking since both the FIN and FOUT metrics remain relatively static throughout. This is in complete contrast to the Djvu package where FIN rises rapidly and FOUT only marginally. The fluctuation in FIN and FOUT values is also evident for the Anno package (Figure 7.7).

The noticeable feature of Table 7.10 is the relatively high values of FOUT compared with FIN throughout. On the basis that, in theory, classes with a high FOUT are easier to modify than classes with a high FIN basis, and from the systems studied so far, this system is certainly the most contrasting in terms of its FIN and FOUT values and presents best opportunity for re-engineering of classes.

Table 7.10 FIN and FOUT per package (DjVu)

Package	Metric (Ver.)	Mean	Median
Djvu	FOUT(V1,V2)	6.21	5
	FOUT (V3)	6.5	5
	FOUT(V4,V5,V6)	6.53	5
	FOUT (V7)	6.97	5
	FOUT (V8)	6.89	5
	FIN (V1,V2)	*10.29*	*7.5*
	FIN (V3,V4)	*13.15*	*10*
	FIN (V5,V6)	*13.23*	*10*
	FIN (V7,V8)	*15.71*	*10.5*
Anno	FOUT (V1,V2)	9.43	4
	FOUT (V3,V4)	7.78	4
	FOUT (V5)	12.86	11
	FOUT (V6)	16.5	11.5

	FOUT (V7)	14.5	3.5
	FOUT (V8)	8	3
	FIN (V1,V2)	*1.14*	*0*
	FIN (V3,V4)	*1.33*	*0*
	FIN (V5)	*7.57*	*5*
	FIN (V6)	*9.17*	*4.5*
	FIN (V7)	*4.67*	*3*
	FIN (V8)	*2.55*	*3*
Toolbar	FOUT (V1)	25	15.5
	FOUT (V2)	14.56	12
	FOUT (V3,V4)	14.89	12
	FOUT (V5)	14.89	12
	FOUT V6,V7,V8)	15.11	12
	FIN (V1)	*4.2*	*1*
	FIN (V2)	*2.33*	*1*
	FIN (V3,V4)	*3.78*	*1*
	FIN (V5)	*3.89*	*1*
	FIN (V6, V7, V8)	*4.11*	*1*

7.4.2.4 The pBeans System

Table 7.11 shows the correlation data for the pBeans system. We considered two packages for this system – pBeans (again a package that shares its name with the system in which it is located) and Data.

We calculated the correlations between FIN and FOUT for 37 classes for pbeans package and 30 classes in the Data package (this is how many classes comprised 80% of the FIN across the whole package). A negative correlation between the FIN and FOUT for the pBeans package is evident; however there is no significant correlation between these two metrics for the Data package. Both sets of correlations are negative but only in some cases are they significant. The data in

Table 7.11 suggests in this system there is a mixture of classes in terms of FIN and FOUT given by the inconsistent pattern of correlations.

Table 7.11 Correlations FIN vs. FOUT (pBeans)

Package	Pearson’s	Kendall’s	Spearman’s
PBeans	-0.064	-0.271*	-0.355*
Data	-0.676**	0.052	0.087

*Correlation is significant at the 0.05 level (1-tailed).
**Correlation is significant at the 0.01 level (1-tailed).

Figure 7.9 shows the values of the FIN and FOUT values for the pBeans package over the versions studied. The peak noticeable for this system was for the StoreException class, which started with a FIN value of 52 and reached 95 by the sixth version. The pBeans package presents an interesting case where the FOUT is significantly larger than FIN in some cases (evident from Figure 7.9). In particular, the class with the high FOUT was called ‘Store’ and comprised 88 methods and 690 LOC. The mean number of methods was just 11 and mean LOC 64. The FOUT for this class was 65 at the final version and the FIN just 30. This illustrates that a class, perhaps crucial to a system, can be one that has a high FOUT but not necessarily a correspondingly high FIN. In contrast to a server class such as GRect described earlier, there may also be ‘client’ classes that use a wide variety of other classes.

Figure 7.10 shows the values of the FIN and FOUT for the Data package. The remarkable feature is the exceptionally low values of FOUT across the classes. One of the explanations for such a low FOUT and high FIN is that many of the classes were ‘descriptor’ classes which many classes would want to use, but equally classes that would not ordinarily use classes themselves. These classes are again server classes that provide a service to other client classes.

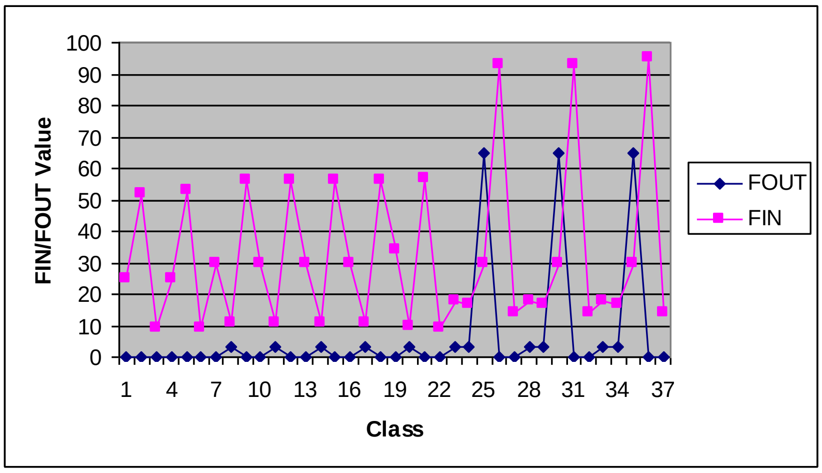

Figure 7.9 FIN/FOUT for the pBeans package

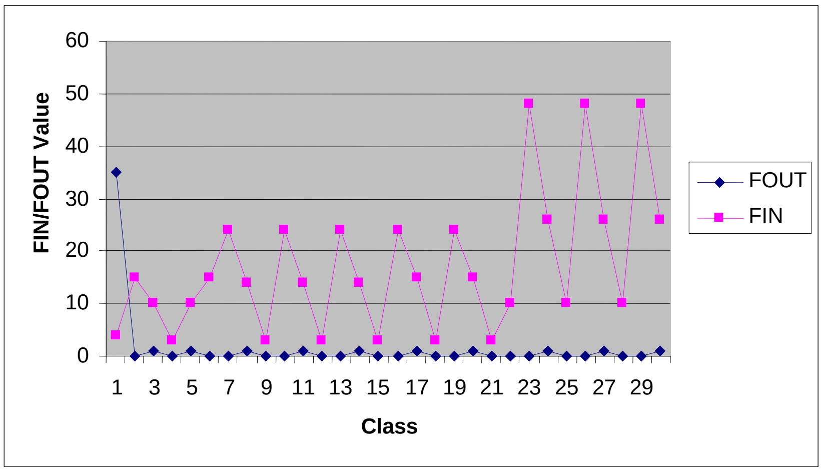

Figure 7.10 FIN/FOUT for the data package

Table 7.12 presents a summary for the FIN and FOUT metrics over the released versions for all classes in the two packages in the pBeans system. The value of the maximum number for FIN over the ten versions is that for the StoreException class in pBeans package and for FieldDescriptor in the Data package. However, the value of the FOUT metrics for these classes, as hinted previously, is zero.

Table 7.12 shows fluctuating values in FIN and FOUT and in contrast to previous systems there appears to be little pattern to these fluctuations. Bearing in mind the inconsistent trends in FIN and FOUT (Figures 7.9 and 7.10), this might have been expected.

Table 7.12 FIN and FOUT per package (pBeans)

Package	Metric (Ver.)	Mean	Median
pBeans	FOUT (V1,V2)	7.21	0
	FOUT (V3,V4,V5)	6.69	0.5
	FOUT (V6)	6.94	1
	FOUT (V7)	5.85	0.5
	FOUT (V8,V9,V10)	9.19	1.5
	FIN (V1,V2)	*8.21*	*4*
	FIN (V3,V4,V5)	*8.56*	*4*
	FIN (V6)	*8.06*	*2*
	FIN (V7)	*8.15*	*6*
	FIN (V8,V9,V10)	*9.58*	*5*
Data	FOUT (V1,V2)	4.53	2
	FOUT (V3,V4,V5)	5.53	2
	FOUT (V6,V7)	6	2
	FOUT (V8,V9,V10)	5.31	0.5
	FIN (V1,V2)	*2.33*	*0*
	FIN (V3,V4,V5)	*2.65*	*0*
	FIN (V6,V7)	*2.42*	*0*
	FIN (V8,V9,V10)	*6.56*	*1.5*

7.4.2.5 The Asterisk System

Table 7.13 shows the correlation data for the Asterisk system. There are three packages considered for this system: Fastagi, Manager and Manager.event. The numbers of the classes that we used to calculate the correlations between FIN and FOUT were 29, 42 and 123 classes for the Fastagi, Manager and Manager.event packages, respectively. There is a negative correlation between the FIN and FOUT for the Fastagi package and a positive correlation between these metrics for the Manager and Manager.event packages over the seven versions of the Asterisk system. Again, there is no consistency amongst the correlation values in terms of their direction.

Table 7.13 Correlations FIN vs. FOUT (Asterisk)

Package	Pearson’s	Kendall’s	Spearman’s
Fastagi	-0.209	-0.276**	-0.326**
Manager	0.025	0.417**	0.471**
Manager.event	0.254**	0.215**	0.244**

**Correlation is significant at the 0.01 level (1-tailed).

Figure 7.11 shows the FIN and FOUT values for the Fastagi package and explains the negative correlations found for this package in Table 7.13. There are exceptionally high values of FIN and correspondingly low values of FOUT. One salient feature of the Fastagi is the number of exception handling classes, each of which has an exceptionally large FIN and low FOUT. The same classes are also relatively small, containing only a few methods. For example, the AGIException class has 2 methods, a value of 109 for FIN and value 0 for FOUT. Figure 7.12 shows the FIN and FOUT values for the Manager package. The values of FIN and FOUT correspond to a greater extent in this package. Again, we see the existence of server classes with a very high FIN but low FOUT. The Fastagi package is an interesting case from a FIN and FOUT point of view – the classes with the highest FIN are all of a certain type – namely exception handling classes. In practice, it makes sense to group these types of classes together, but this did not seem to be a feature of any of the other systems studied.

Figure 7.13 shows the FIN and FOUT values for the Manager.event package. The relatively high FIN values of 28 belong to a class called ManagerEvent. Again the high value of FIN and relatively low value of FOUT for this class makes sense since many classes would want to access this class for the critical functionality it offers (that of event handling).

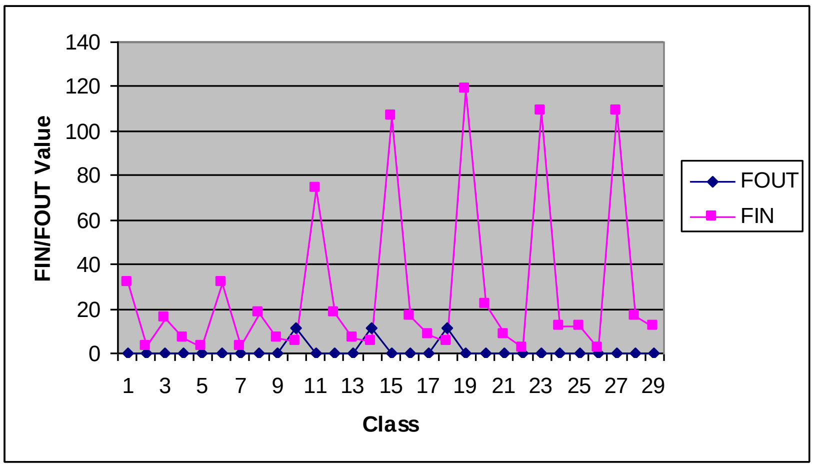

Figure 7.11 FIN/FOUT for the Fastagi package

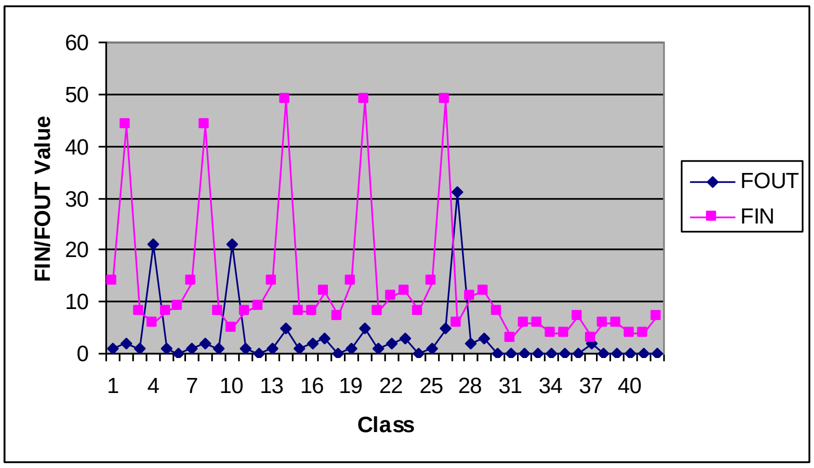

Figure 7.12 FIN/FOUT for the Manager package

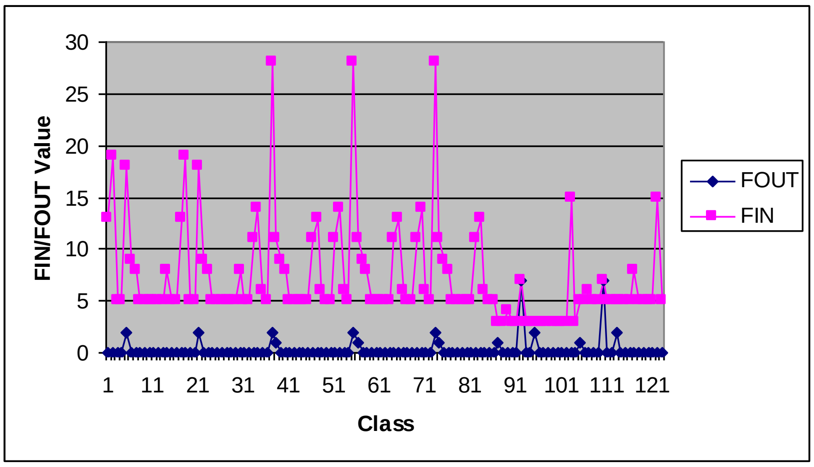

Figure 7.13 FIN/FOUT for the Manager.event package

From the five systems studied, we see that there are certain classes with a low FIN and high FOUT, but more frequent is the occurrence of a class with the opposite characteristics (high FIN, low FOUT). Table 7.14 presents a summary for the FIN and FOUT metrics over the released versions for the three packages.

Table 7.14 FIN and FOUT per package (Asterisk)

Package	Metric (Ver.)	Mean	Median
Fastagi	FOUT (V1,V2)	1.83	0
	FOUT(V3,V4,V5)	2	0
	FOUT (V6,V7)	2.21	0
	FIN (V1,V2)	*3*	*1*
	FIN (V3)	*5.17*	*1*
	FIN (V4)	*6.08*	*0*
	FIN (V5)	*6.76*	*0*
	FIN (V6,V7)	*7.21*	*0*
Manager	FOUT (V1,V2)	3.67	0
	FOUT(V3,V4,V5)	4.23	1
	FOUT (V6)	0.64	0
	FOUT (V7)	6.17	0
	FIN (V1,V2)	*3.55*	*0*
	FIN (V3,V4,V5)	*3.26*	*0*
	FIN (V6,V7)	*3*	*3*
Manager. Event	FOUT (V1,V2)	0.04	0
	FOUT V3,V4,V5)	0.05	0
	FOUT (V6,V7)	0.15	0
	FIN (V1,V2)	*3.18*	*1*
	FIN (V3,V4,V5)	*2.99*	*1*
	FIN (V6)	*1.83*	*1*
	FIN (V7)	*3.48*	*3*

The maximum value for FIN over the seven versions is actually the FIN for the AgiException class in Fastagi, for the Channel class in the Manager package and for the ManagerEvent class in the Manager.event package. However, the values of the FOUT metrics for these classes are trivially small. There is a wide fluctuation in mean values for the two metrics over the versions studied. However, in keeping with most of the previous systems – the median values suggest that there is no significant change in terms of these two metrics.

7.5 Summary

In this chapter, we have explored the 80/20 in four OSS and six coupling metrics. The key findings were that, to a limited extent, FIN and FOUT metrics had a greater propensity for that rule than others. Moreover, these two metrics have a complementary relationship. We also found that many of the other metrics had few, if any, such relationships. The RFC was typical in this sense; no 80/20 relationship was found in any of the systems or any version in those systems. Finally, an 80/20 rule did not seem to worsen as a system evolved. Because of these results, we investigated the relationship between the FIN and FOUT in five OSS. The key finding was that for most of the systems there is a correlation between the changes in the FIN and FOUT. This correlation was negative for some packages but positive for most of the packages – this informed our interpretation of the two metrics. We also asked two significant questions. First, what is the nature and characteristics of classes exhibiting the highest FIN values? Second, do FIN and FOUT increase in corresponding and consistent amounts over time? Our analysis revealed a wide range of traits in the classes to explain high and low levels of FIN and FOUT. We found evidence of certain ‘key’ classes with high FIN and FOUT; we also found evidence of classes with just high FIN (‘server’-type) classes. In certain cases, the size of a class revealed its purpose as much as the values of FIN and FOUT. Finally, evolutionary aspects also showed evidence of a range of coinciding ‘similar directions’ of evolution; in other cases, we found unilateral and independent evolution with respect to the two metrics studied.

CHAPTER 8. CONCLUSIONS, REFLECTIONS AND FURTHER WORK

8.1 Overview

This chapter draws together the research findings to present an understanding of coupling between classes in Java OSS. The chapter starts with an overall summary of the research and relates the key research findings with the conclusions. Contributions from the research findings are discussed. Finally, the research limitations are presented together with potential future research directions.

8.2 Research Summary

The objectives of this Thesis, originally stated in Chapter 1, were:

1. To give a project manager an idea of future maintenance or refactoring opportunities by understanding changes in a system through the analysis of its packages and finding the link between these changes and the refactoring.

2. To investigate characteristics of classes removed from systems during their evolution from the perspective of their coupling to other classes, their size compared to other classes and their change trends before they were removed.

3. To discover the relationship between changes in the coupling metrics over the releases of a system and the different time periods between these releases, on the one hand, and the relationship between these changes and the code warnings, on the other hand.

4. To explore whether an 80/20 rule exists in Java from coupling metrics over multiple versions of OSS.

5. To investigate the characteristics of classes that show the highest value of incoming coupling metrics. In particular, to address the issue of potential re-engineering and to view coupling as a key contributor to the decision on whether or not and when to re-engineer (classes) over the lifetime of a system.

To address the previous objectives, we started our investigation of coupling by conducting a comprehensive literature review (Chapter 2) of previous work reported on coupling, how it was used in practice and what implications it may have on system maintainability. This was followed by providing a detailed description of the research methodology implemented in the Thesis, including the systems and the software metrics used in the study (Chapter 3).

In Chapter 4, we investigated how Velocity system evolved at the package level. The trends in changes of nine versions of Velocity were explored through three research questions. An interesting ‘peak and trough’ effect trend existed in specific versions of the system. A contrast was found between those regular changes and those associated with refactoring activity.

In Chapter 5, removed classes were investigated in four Java systems. Five coupling metrics were collected from these four Java open-source systems using the JHawk tool. By investigating the influence of the extent coupling and the class size on the removal of the classes from the system, we found that FIN and FOUT tended to be comparatively small for these classes; however, little evidence that the size influenced removed classes was found. Finally, changes for most of the classes were made in early versions before the classes were removed relatively later on.

In Chapter 6, trends in coupling in five Java systems were investigated. Again, five coupling metrics and the warning for each version were collected from the five Java OSS. Investigation of the five systems revealed that adding coupling may have advantageous effects on a system. Moreover, it seems that addition of coupling in new functionality through addition of packages could result in fewer warnings than adding functionality to existing codes.

The 80/20 rule in four OSSs and six metrics was explored in Chapter 7. FIN and FOUT had a larger leaning toward that rule than others. High use of these two metrics seemed to eliminate the use of other types of coupling. Consequently, the relationship between FIN and FOUT was investigated. For most of the studied systems there was a correlation between the changes in the FIN and FOUT. Our analysis showed a wide range of traits in the classes to explain both high and low levels of FIN and FOUT. We also found evidence of certain 'key' classes with both high FIN and FOUT and 'client' and 'server'-type classes with just high FOUT and FIN, respectively.

Based on the results of empirical studies presented throughout the Thesis, we therefore feel that all the research objectives have been satisfied. The trends in coupling in the package level in the evolution of Java OSS have presented interesting characteristics. We therefore assert that the Thesis informs our empirical understanding of coupling features of OO from an evolutionary perspective.

8.3 Research Contributions

The research described in this Thesis relates to areas of software evolution, coupling metrics and potential fault analysis (through warnings) and the use of OSS. As stated in Chapter 1, this study contributes to an empirical body of knowledge on coupling and longitudinal analysis, of which more studies are recommended (Kemerer and Slaughter, 1999a; 1999b). Few empirical studies investigating coupling from the perspective of evolution in OSS can be found in the current literature. Equally, empirical evidence exists to suggest that research on software evolution is conducted inadequately. Kemerer and Slaughter (1999a) state that there is a need for further empirical studies in the software evolution.

While there have been multiple studies of coupling both in the procedural and OO arena, the results that we report in this Thesis suggest that coupling is a multi-faceted, multi-dimensional and more complex feature of a system than we may have appreciated in the past.

The main contribution of this Thesis can be seen in the light of few research stands.

- An appreciation of trends of maintenance changes can help predict future changes in the maintenance practice. This may also help a project manager to estimate the likely maintenance effort needed to keep the project working properly. Based on the trends of activity changes, developers can take preventive action for further system maintenance and/or refactorings. The findings of Chapter 4 suggested that maintenance changes follow an interesting 'peak and trough' effect trend in specific versions of the system. These trends corresponded with empirical evidence in refactoring data for the same system. This result suggests a contrasting motivation between *regular* maintenance practice and that of refactoring. In other words, refactoring might be applied after a burst of regular change activity, rather than consistently.

- Since few empirical studies have analysed coupling from an evolutionary respective, we believe the methodological approaches adopted for data collection and analysis in this Thesis can help inform future empirical studies on coupling and its evolution. The Thesis therefore makes a contribution to our understanding of how coupling evolves and where the majority of maintenance changes are applied. In Chapter 5, results showed a strong tendency for classes with low FIN and FOUT to be candidates for removal. Evidence was also found of class types with high imported package and external call functionality being removed. The research addressed an area that is often overlooked in the study of evolving systems, notably the characteristics and features of classes that disappear from a system.

- The findings in Chapter 6 recommended that addition of coupling might have beneficial effects on a system. It also seemed that there was a coupling trade-off between coupling types, in particular that between the uses of coupling through imported packages and the introduction of ‘internal-to-the-package’ coupling.

- Chapter 7 provided an explanation of the composition and existence of ‘key’ and ‘server’-type classes as well as for disproportionate increases in each of the two metrics (FIN and FOUT) over time. The research presented suggests that there is no such thing as completely common trends in systems as far as coupling is concerned and that there are multiple reasons why classes may be highly or minimally coupled through FIN, FOUT or a combination of both. Equally, there are other class characteristics that play a crucial part, such as size, and associated with that, the level and type of functionality. The ‘key’ and ‘server’ classes feature in the evolution of a system and its functioning.

8.4 Personal Achievement

There are many things that have been achieved over the course of the research in this Thesis. First of all, the process of conducting research needs an advancement of the researcher’s knowledge in the studied area, and that was achieved by reading more about this subject and searching the literature for related papers. Moreover, research is usually done under time pressure, so time management is a key aspect of research and a good help in meeting deadlines. Additionally, successful research needs good collaboration and communication with other researchers in the same study area, and that was achieved by communicating with colleagues within the university or by attending conferences.

The research in this Thesis required a certain amount of data collection and statistical analysis. This helped in improving an awareness of data collection and suitability of statistical tests for a set of data. Over the course of this Thesis, it was evident that completing a PhD is merely a learning process for further scientific research. Finally, writing this research has raised the ability to read and write technically and critically and has improved personal skills as a researcher.

8.5 Research Limitations and Future Work

One threat to the validity of the study is that we have only used seven OSSs as part of our study. While that provides a cross-sectional view of systems, we accept that this limited number threatens the generalisability of the results.

In the Thesis analysis, there was a focus on coupling and class size; however, many other features of classes could have been used as a basis, for example, their cohesion or their position in the inheritance hierarchy. We leave such analyses for future work, however.

Another threat to the validity of the study is that we have used warnings as a basis of our analysis in Chapter 6 and not actual faults or complementary techniques. However, we feel that it is better to be 'fore-warned' and therefore 'fore-armed' of potential problems and to analyse that data, than to analyse data in a post-fault sense.

Finally, a threat to the study validity is that we have only collected six coupling metrics from a wide range of available coupling metrics in the literature. We defend this choice on the basis that these six give a set of metrics that allow different levels of code and design abstraction to be analysed and compared; this is a key objective of the study.

In terms of future work, we view the research as a starting point for further replicated studies and for an in-depth and generalised analysis of coupling/refactoring, both inter- and intra-package. Consequently, we aim to extend our analysis to more systems and more versions in the spirit of Girba et al. (2005).

From a refactoring perspective, it would be useful to investigate the link that removal of classes may have with refactoring, and specifically with respect to package refactoring. We also expect to investigate the potential for refactoring the code as a result of that warning data. Future work will focus on first exploring the opportunities and effects of applying refactorings to classes that appear to have a high FIN and FOUT; it will also consider a finer-grained analysis of the different types of coupling inherent in the classes studied and a coupling analysis ‘normalised’ by class size.

We would encourage further empirical studies into coupling and particularly evolutionary-based studies to refute, support or complement the results in this Thesis; to that end, all the data used in this Thesis can be obtained for replication purposes and other analyses on request from the author.

REFERENCES

Abreu, F. and Carapuca, R. (1994) Object-oriented software engineering: measurement and controlling the development process, *Revised version: Originally published in the Proceedings of the 4th International Conference on Software Quality*. McLean, VA.

Advani, D., Hassoun, Y. and Counsell, S. (2005) Refactoring trends across N versions of N Java open source systems: an empirical study, *Technical Report* BBKCS-05-03-01. Birkbeck, University of London: London, UK.

Advani, D., Hassoun, Y. and Counsell S. (2006) Extracting refactoring trends from open-source software and a possible solution to the related refactoring conundrum, *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, Dijon, France, April 23-27, 2006.

Akiyama, F. (1971) An example of software system debugging, *Proceedings of the IFIP Congress,* pp. 353-379.

Arisholm, E. and Briand, L.C. (2006) Predicting fault-prone components in a Java legacy system, *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering,* pp. 8-17.

Arisholm, E., Briand, L.C. and Foyen, S. (2004) Dynamic coupling measurement for object-oriented software, *IEEE Transactions on Software Engineering*, 30(8) pp. 491-506.

AspectJ, (2005) Available at (http://www.eclipse.org/aspectj) [Accessed 10th December 2009].

Bartsch, M. and Harrison, R. (2006a) A coupling framework for AspectJ, *Proceedings of the 10th International Conference on Evaluation and Assessment in Software Engineering*, Keele, UK, April 2006.

Bartsch, M. and Harrison, R. (2006b) An evaluation of coupling measures for AspectJ, *Workshop LATE'06, in conjunction with Aspect-Oriented Software Development Conference (AOSD'06)*, Bonn, Germany, March 20, ACM Press.

Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgard, S. and Zelkowitz, M.V. (1996a) The empirical investigation of perspective-based reading, *Empirical Software Engineering: An International Journal*, 1(2), pp. 133–164.

Basili, V.R., Briand, L.C. and Melo, W.L. (1996b) A validation of object-oriented design metrics as quality indicators, *IEEE Transaction on Software Engineering*, 22(10), pp. 751-761.

Beck, K. (1999) *Extreme programming explained: embrace change.* Addison-Wesley.

Belady, L.A. and Lehman, M.M. (1976) A model of large program development, *IBM System Journal*, 15(3), pp. 225-252.

Bennett, S., McRobb, S. and Farmer, R. (2002) *Object-oriented systems analysis and design using UML.* 2nd ed. London: McGraw Hill.

Bennett, K. (1996) Software evolution: past, present and future, *Information and Software Technology*, 38, pp. 673-680.

Bieman, J. M., Straw, G., Wang, H., Munger, P. and Alexander, R. (2003) Design patterns and change proneness: an examination of five evolving systems, *Proceedings of the 9th International Software Metrics Symposium*, Sydney, Australia, pp. 40-49.

Bieman, J.M., Jain, D. and Yang, J.Y. (2001) OO design patterns, design structure, and program changes: an industrial case study, *Proceedings of the International Conference on Software Maintenance*, Florida, Italy, pp. 580-589.

Black, S. (2001) Computing ripple effect for software maintenance, *Journal of Software Maintenance and Evolution: Research and Practice*, 13, pp. 263-279.

Briand, L., Devanbu, P. and Melo, W. (1997) An investigation into coupling measures for C++, *Proceedings of the 19th International Conference on Software Engineering*, Boston, USA, pp. 412-421.

Briand, L., Daly, J., Porter, V. and Wust, J. (1998) Predicting fault-prone classes based on design measures in object-oriented systems, *Proceedings of the 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 4-7 November 1998, pp 334-343.

Briand, L.C., Arisholm, F., Counsell, S., Houdek, F. and Thevenod-Foss, P. (1999a) Empirical studies of object-oriented artifacts, methods and processes: state of the art and future directions, *Empirical Software Engineering: An International Journal*, 4(4), pp. 387-404.

Briand, L., Daly, J. and Wust, J. (1999b) A unified framework for coupling measurement in object-oriented systems, *IEEE Transactions on Software Engineering*, 25(1), pp. 91-121.

Briand, L., Wust, J., Ikonomovski, S. and Lounis, H. (1999c) Investigating quality factors in object-oriented designs: an industrial case study, *Proceedings of the International Conference on Software Engineering*, Florida, USA, pp. 345-354.

Burd, E. and Munro, M. (2000) Using evolution to evaluate reverse engineering technologies: mapping the process of software change, *Journal of Systems and Software*, 53 (1), pp. 43-51.

Cartwright, M. and Shepperd, M. (2000) An empirical investigation of an object-oriented (OO) system, *IEEE Transactions on Software Engineering*, 26, pp. 786-796.

Ceccato, M. and Tonella, P. (2004) Measuring the effects of software aspectization, *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004).* Delft, The Netherlands.

Chidamber, S.R. and Kemerer, C.F. (1994) A metrics suite for object oriented design, *IEEE Transactions on Software Engineering,* 20(6), pp. 476-493.

Chikofsky, E.J. and Cross, J.H. (1990) Reverse engineering and design recovery: a taxonomy, *IEEE Software,* 7(1), pp.13–17.

Counsell, S., Mubarak, A. and Hierons, R. (2010) An evolutionary study of Fan-in and Fan-out metrics in OSS. *Proceedings of the 4th International Conference on Research Challenges in Information Science (RCIS 2010),* Nice, France, 2010.

Counsell, S. (2008) Do student developers differ from industrial developers? *Proceedings Information Technology Interfaces (ITI) Conference*, Dubrovnik, Croatia, June 2008.

Counsell, S., Hierons, R.M., Najjar, R., Loizou, G. and Hassoun, Y. (2006) The effectiveness of refactoring based on a compatibility testing taxonomy and a dependency graph, *Proceedings of Academic and Industrial Conference (TAIC PART),* Windsor, UK, August 2006, pp. 181-190. IEEE Computer Society Press.

Creswell, J. W. (2003) *Research design, qualitative, quantitative and mixed methods approaches*. 2nd ed. London, Sage.

Demeyer, S., Ducasse, S. and Nierstrasz, O. (2000) Finding refactorings via change metrics, *Proceeding of OOPSLA '00: Proc of the 15th ACM SIGPLAN OOPSLA*, Minneapolis, Minnesota, United States, pp.166–177. ACM Press, New York, NY, USA.

Dinh-Trong, T. and Bieman, J. (2004) Open source software development: a case study of FreeBSD, *Proceedings of the 10th IEEE International Symposium on Software Metrics,* Chicago, USA, pp. 96-105.

Ducasse, S., Lanza, M. and Ponisio, L. (2005) Butterflies: a visual approach to characterize packages, *Proceedings of the 11th International Software Metrics Symposium*, Como, Italy, pp. 7-16.

Ducasse, S., Lanza, M. and Ponisio, L. (2004) A top-down program comprehension strategy for packages. *Technology Report IAM-04-007, University of Berne, Institute of Applied Mathematics and Computer Sciences*.

Dvorak, J. (1994) Conceptual entropy and its effect on class hierarchies, *IEEE Computer*, 27(6), pp. 59-63.

El Emam, K., Benlarbi,S., Goel, N. and Rai, S. (2001)The confounding effect of class size on the validity of object-oriented metrics, *IEEE Transactions on Software Engineering*, 27, pp. 630-650.

English, M., Buckley, J. and Cahill, T. (2007) Fine-grained software metrics in practice, *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM "07)*, Madrid, Spain, pp. 295-304.

Fenton, N.E. and Pfleeger, S.L. (2002) *Software metrics: a rigorous and practical approach*. London, UK: International Thomson Computer Press.

Fenton, N and Neil, M (1998) Software metrics: successes, failures and new directions, *Journal of Systems and Software*, 47(2-3), pp. 149-157.

Field, A. (2005) *Discovering statistics using SPSS.* 2nd ed. London, Sage.

Findbug, (2008) Available at (http://findbugs.sourceforge.net/). [Accessed 15th March 2008]

Foote, B. and Opdyke, W. (1995) *Life cycle and refactoring patterns that support evolution and reuse. Pattern languages of programs*. Addison-Wesley.

Fowler, M. (1999) *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design patterns: elements of reusable object-oriented software.* Massachusetts: Addison- Wesley.

Gilb, T. (1976) *Software metrics.* Cambridge, MA: Chartwell-Bratt.

Girba, T., Ducasse, S. (2006) Modeling history to analyse software evolution. *Journal of Software Maintenance and Evolution*, 18(3), pp. 207-236.

Girba, T., Lanza, M. and Ducasse, S. (2005) Characterizing the evolution of class hierarchies, *Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* Manchester, UK, pp. 2-11.

Granja-Alvarez, J. C. (2004) New technologies, software maintenance: a case study, *Proceedings of the Ninth IEEE Workshop on Empirical Studies of Software Maintenance*. Chicago, USA.

Hall, T., Rainer, A. and Jagielska, D. (2005) Using software development progress data to understand threats to project outcomes, *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, Como, Italy, 10 pages.

Harrison, R., Counsell, S. and Nithi, R. (1998) An evaluation of the MOOD set of object-oriented software metrics, *IEEE Transactions on Software Engineering*, 24(6), pp. 491-496.

Harrison, R., Counsell, S. and Nithi, R. (1997) An overview of object-oriented design metrics, *Proceedings of the conference on Software Technology and Engineering Practice (STEP),* IEEE Press, pp. 230-237.

Hautus, E. (2002) Improving Java software through package structure analysis, *Proceedings of the Sixth IASTED International Conference Software Engineering and Applications*, Cambridge, USA.

Henry, S.M and Kafura, D.G. (1981) Software structure metrics based on information flow, *IEEE Transactions on Software Engineering,* 7(5), pp. 510-518.

Jajodia, S. and Kogan, B. (1990) Integrating an object-oriented data model with multilevel security, *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, pp. 76-85.

JHawk tool (2008) Available at: (http://www.virtualmachinery.com/jhawkprod.htm) [Accessed 13th January 2008]

Johnson, R.E. and Foote, B. (1988) Designing reusable classes, *Journal of Object-Oriented Programming*, 1(2), pp. 22-35.

Kajko-Mattsson, M., Forssander, S. and Olsson, U. (2001) Corrective maintenance maturity model (CM^3): maintainer's education and training, *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, pp.610-619.

Kemerer, C.F. and Slaughter, S. (1999a) Need for more longitudinal studies of software maintenance. *Empirical Software Engineering: An International Journal*, 2(2), pp. 109-118.

Kemerer, C.F. and Slaughter, S. (1999b) An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4), pp. 493-509.

Kerievsky, J. (2002) *Refactoring to patterns*. Addison Wesley.

Kitchenham, B.A., Pfleeger, S.L. and Fenton, N.E. (1995a) Towards a framework for software measurement validation, *IEEE Transactions on Software Engineering*, 21(12), pp. 929-944.

Kitchenham, B., Pickard, L. and Pfleeger, S.L. (1995b) Case studies for method and tool evaluation. *IEEE Software*, 12 (4), pp. 52-62.

Lehman, M. (1998) Understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1, pp. 213-221.

Li, W. and Henry, S. (1993) Object-oriented metrics that predict maintainability, *The Journal of Systems and Software*, 23 (2), pp. 111-122.

Longstreet D.H. (1990) *Software maintenance and computers.* IEEE Computer Society Press U.S.

Lorenz, M. and Kidd, I. (1994*) Object-oriented software engineering metrics*. Prentics-Hall Englwood Cliff, NJ.

McDermid, J. A. (1991) *Software engineering reference book*. Oxford: Butterworth-Heinemann Ltd.

Mens, T., Ramil, J. and Godfrey, M. (2004) Analyzing the evolution of large-scale software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6), pp. 363-365.

Mens, T. and Tourwe, T. (2004) A survey of software refactoring, *IEEE Transactions on Software Engineering*, 30(2), pp. 26-139.

Mens, T. and van Deursen, A. (2003) Refactoring: emerging trends and open problems, *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*. University of Waterloo.

Mubarak, A., Counsell, S. and Hierons, R. (2009) Does an 80:20 rule apply to Java coupling? *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*, Keele, UK.

Mubarak, A., Counsell, S. and Hierons, R. (2008a) An empirical study of "removed" classes in Java open-source, *Proceedings of the 4th International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 08)*.

Mubarak, A., Counsell, S. and Hierons, R. (2008b) Empirical observations on coupling, code warnings and versions in Java open-source, *Proceedings of the3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2008),* Brno, Czech Republic.

Mubarak, A., Counsell, S., Hierons, R. and Hassoun, Y. (2007) Package evolvability and its relationship with refactoring, *Proceedings of the 3rd International ERCIM Symposium on Software Evolution*, Paris, France.

Mubarak, A. Counsell, S., Hierons, R. and Hassoun, Y. (2007) Package evolvability and its relationship with refactoring, *Electronic Communication of the European Association of Software Science and Technology*, 8.

Najjar, R., Counsell, S., Loizou, G. and Mannock, K. (2003) The role of constructors in the context of refactoring object-oriented software, *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR '03)*, Benevento, Italy, pp. 111-120.

Opdyke, W.F. and Johnson, R.E. (1990) Refactoring: an aid in designing application framework and evolving object-oriented systems, *Proceedings of the Symposium on Object-Oriented Programming, Emphasizing Practical Applications (SOOPPA ‟90)*, Poughkeepsie, NY, pp. 145-161.

Opdyke, W. (1992) Refactoring object-oriented frameworks, Ph.D. Thesis, Univ. of Illinois.

Perry, D.E. (1994) Dimensions of software evolution, *Proceedings of the International Conference on Software Maintenance*, Victoria, BC, Canada, pp. 296-303.

Pillai, K. (1996) The fountain model and its impact on project schedule. *Software Engineering Notes*, 21(2), pp. 32-38.

Robson, C. (2002) *Real world research: a resource for social scientists and practitioner-researchers.* 2nd ed. Oxford, Blackwell Publishers Ltd.

Shepperd, M.J. (1995) *Foundations of software measurement*. Hertfordshire, UK: Prentice Hall International.

Smith, N., Capiluppi, A. and Fernandez-Ramil, J. (2006) Agent-based simulation of open source evolution, *Journal of Software Process - Improvement and Practice*, 11(4), pp. 423-434.

Sommerville, I. (1996) *Software engineering.* 5th ed. Wokingham, England, Addison-Wesley.

Stevens, W., Myers, G. and Constantine, L. (1974) Structured Design, *IBM Systems Journal*, 13, pp. 60-73.

Tourwe, T. and Mens, T. (2003) Identifying refactoring opportunities using logic meta programming, *Proceedings of the 7th European Conference on Software Maintenance and Re-Engineering*, Benevento, Italy, pp. 91-100.

van Deursen, A. and Moonen, L. (2002) The video store revisited – thoughts on refactoring and testing, *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP*. Sardinia, Italy, pp. 71-76.

Wasserman, A. (1996) Toward a discipline of software engineering, *IEEE Software*, 13(6), pp. 23-31.

Wheeldon, R. and Counsell, S. (2003) Power law distributions in class relationships. *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, pp. 45-54.

Wilde, N. and Huitt, R. (1992) Maintenance support for object-oriented programs, *IEEE Transactions on Software Engineering*, 18(12), pp. 1038-1044.

Williams, B.J. and Carver J.C. (2007) Characterizing software architecture changes: an initial study, *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, pp. 410-419.

Wohlin, C. Runeson, P., Host, M., Ohlsson, M., Regnell, B. and Wesslen, A. (2000) *Experimentation in software engineering: an introduction.* Boston/ Dordrecht/ London, Kluwer Academic Publishers.

Wu, J., Holt, R. and Hassan, A. (2007) Empirical evidence for SOC dynamics in software evolution, *Proceedings of the 24th International Conference on Software Maintenance*, Paris, France, pp. 244-254.

Zhang, J. Lin, Y. and Gray J. (2005) Generic and domain-specific model refactoring using a model transformation engine, *Springer Berlin Heidelberg*, pp. 199-217.

Zheng, J, Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. and Vouk, M. (2006) On the value of static analysis for fault detection in software, *IEEE Transactions on Software Engineering*, 32(4), pp. 240-253.

Zimmermann, T., Weißgerber, P., Diehl, S. and Zeller, A. (2005) Mining version histories to guide software changes, *IEEE Transactions on Software Engineering*, 31(6), pp. 429-445.

GLOSSARY OF SOFTWARE ENGINEERING TERMS

The terms define below are widely used in software engineering. The purpose of this glossary is to explicitly indicate what we mean by each term in this Thesis and avoid any confusion by the reader.

Abstraction

The concept of abstraction from the perspective of OO is a process that involves identifying only crucial aspects of a problem and ignoring the non-essential information and detail.

Attribute

Attributes are data fields defined in a class to store information about each instance/object of that class.

Bad Smells

Bad smells in code are strong indicators of problems somewhere in the code that offer opportunities for refactoring.

Class

A class is a unit of code from which instance objects are created and defines a set of attributes and methods for those objects.

Cohesion

Cohesion is the extent of class components working together to perform one single and precise task. Cohesion increases class comprehensibility and eases modification.

Coupling

Coupling in OO is a measure of inter-dependency between classes. High coupling shows a strong dependency which is undesirable from a complexity perspective.

Design Patterns

Design patterns are recurring solutions to software design problems that are observed or discovered repeatedly in real-world application development environments (Gamma et al. 1995)

Fan-in Metrics

Fan-in metrics is the number of functions that call a particular function. A function with a high Fan-in means that many other functions use this function.

Fan-out Metrics

Fan-out metrics is the number of functions a function calls.

Inheritance

Inheritance is a mechanism used in OO which provides the ability to define a new class using methods an attributes of an existing class and adding its own specific methods and attributes. The newly added class is then called subclass and the existing class is called superclass.

Java Package

A package in Java is a namespace used to organise class files. This is done by creating a directory, putting all classes with related functionally in that directory and giving it a sensible name to clearly represent the functionality of those classes. The directory in which all classes exist is called a package.

Lines of Code (LOC) Metric

LOC measures lines of code in a system or a class which may or may not include comments and/or blank lines.

Message Passing Coupling (MPC) Metric

MPC metric measures the total number of method calls in the methods of a class to methods of other classes. In other words, it measures the dependency of methods of a class to the methods of other classes.

Method

A method is a member function in a class consisting of a set of statements which may have a set of arguments and may have a return type. Methods are used to provide overall class behaviour.

Move Field (MF) Refactoring

MF refactoring moves a field from a class to another, in which it is used more than the class it is defined (Fowler, 1999). .

Number of Attributes (NOA) Metric

NOA metric measures the total number of local variables plus the total number of class variables (including *public, private and protected*) in a class.

Number of External Methods Called (EXT) Metrics

The more external methods that a class calls the more tightly bound that class is to other classes.

Number of Methods (NOM) Metric

NOM metric measures the total number of methods in a class.

PACK Metrics

PACK metrics measures the number of imported packages.

Refactoring

Refactoring is the process of changing internal behaviour of a system, to make it easy to understand and change, while preserving its external behaviour.

Rename Field (RF) Refactoring

RF refactoring is concerned with changing the name of a field to clearly state its purpose (Fowler, 1999).

Rename Method (RM) Refactoring

RM refactoring is concerned with changing the name of a method to clearly state its purpose (Fowler, 1999).

Response for a Class (RFC) Metrics

This metric is the same as that defined by Chidamber and Kemerer (1994) and measures the response set of a class. The RFC is defined as the set of methods that can be potentially executed in response to a message received by an object of that class.

Software Metrics

Software metrics are measures of characteristics of a software project, product or process.

Superclass

A superclass is a class which contains all the common features (methods and attributes) to be inherited by a set of classes and serves as an ancestor for those classes. The classes inheriting those common features add their own specific features so that more specific objects of the superclass can be created.

Subclass

A subclass is a class which inherits from another class or implements an interface.

Warning

The term warning in the context of this Thesis indicates to the problems embedded in a system which may potentially lead to a fault (FindBugs, 2008).

APPENDIX A: DATA USED IN ANALYSING THE MAINTENANCE CHANGES

The data analysed in Chapter 4 was held in Excel spreadsheets. The data for each package is categorised in several columns, and each column contains the changes that occurred to the packages since the previous version. These tables are as follows.

Note: for all the following tables, when the cell is empty that means the package did not exist for this version.

Table A.1 was used in analysing the first research question. It was created by taking the column that includes the number of the added classes for each package across the nine versions.

Table A.1 The number of the classes added to each package (Velocity)

Version No.\ Package name	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
ant		55	0	1	33	0	6	1	23
apisupport		46	0	0	28	2	0	10	
applet	0	0	0	0	0	0	0	0	0
autoupdate	0	18	0	0	21	0	0	14	8
beans	4	10	0	0	6	0	9	1	3
classfile				0		0	0	1	0
clazz	3	2	1	0	2	0	1	15	2
core	81	162	5	1	316	7	92	131	217
debuggercore	12	105	0	0	121	3	0	23	27
debuggerjpda	1	7	0	0	329	0	0	7	7
debuggertools	0	3	0		4	0	0		
diff				0					1
editor	176	55	0	0	71	1	58	54	97
extbrowser			0	0	19	0	0	1	12
form	37	87	1	0	91	1	0	64	76
html	1	3	0	0	0	0	0	1	0
httpserver	5	3	0	0	2	0	0	9	2
i18n	25	60	0	0	0	0	8	1	4
icebrowser	0	7	0						
image	1	6	0	0	3	0	0	0	1

j2eeserver				0		0	0	47	3
jarpackager	23	43	0	0	21	0	0	111	13
java	50	207	0	0	41	1	35	16	140
javacvs			0	0	153	0	0	64	13
javadoc	0	34	2	3	22	1	0	9	17
jndi			1	0	1	3	0	21	0
nbbuild	18	4	0	2	35	1	0	8	27
objectbrowser	0	0	0						
openide	78	80	2	1	294	13	81	115	148
openidex	10	1	0	0	45	0	0	23	0
projects	8	21	0	0	9	1	7	0	24
properties	20	7	0	0	3	0	0	27	19
rmi			1	0	20	0	0	14	
schema2beans				0					38
scripting		6	0	0	6	0	0	0	
text	3	0	0	0	3	0	0	0	0
tomcatint				0		0	0	52	15
ui				1					81
usersguide			0	0	0	0	0	7	0
utilities	16	10	0	0	14	2	0	2	6
vcscore		62	2	1	146	3	0	50	83
vcscvs	52	7	1	0	4	0	0	0	1
vcsgeneric			1	1	53	6	0	18	57
xml				0					157
web	164	5	0	0	116	0	0	357	64
SUM	788	1116	17	11	2032	45	297	1274	1386
The Max Inc	176	207	5	3	329	13	92	357	217
Package name	editor	java	core	javadoc	debugg-erjpda	openide	core	web	core

Table A.2 was used in analysing the second research question. It was created by taking the column that indicates to the maximum increase in the number of lines of code for each package across the nine versions.

Table A.02 The max. increase in the LOC for each package (Velocity)

Version No.\ Package name	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
ant		105	0	54	131	9	0	36	69
apisupport		66	0	0	42	37	0	94	
applet	29	8	0	0	0	0	0	2	36
autoupdate	13	180	10	16	128	44	34	74	174
beans	137	138	2	13	76	2	0	94	45

classfile				0		0	31	8	3
clazz	16	75	2	17	110	0	0	30	15
core	547	380	51	36	614	103	71	1854	453
debuggercore	106	56	18	0	225	54	35	56	329
debuggerjpda	138	149	8	0	160	15	7	64	77
debuggertools	71	52	0		82	3	0		
diff				165					27
editor	265	145	60	12	498	60	1	258	388
extbrowser			0	0	73	3	0	17	5
form	198	686	30	3	609	89	14	352	591
html	7	16	0	0	3	1	0	90	10
httpserver	43	93	0	0	25	1	0	56	18
i18n	182	88	1	0	57	4	14	24	12
icebrowser	0	112	0						
image	62	185	0	0	14	0	0	102	26
j2eeserver				0		14	0	35	136
jarpackager	213	223	13	7	150	30	22	86	74
java	321	184	21	0	413	22	29	125	387
javacvs			174	320	216	33	2	181	505
javadoc	116	146	11	41	76	20	0	167	29
jndi			44	0	57	91	7	156	0
nbbuild	0	27	0	30	93	52	0	305	203
objectbrowser	2	1	0						
openide	325	206	60	47	698	56	47	351	159
openidex	0	16	0	0	8	9	5	0	3
projects	87	446	5	0	121	57	30	49	167
properties	120	54	7	2	32	11	0	91	24
rmi			41	0	159	14	8	20	
schema2beans				0					1236
scripting		198	0	0	79	3	0	52	
text	52	29	0	0	3	2	0	2	4
tomcatint				0		0	0	574	137
ui				31					84
usersguide			0	0	1	0	0	51	3
utilities	233	175	0	0	63	35	0	43	38
vcscore		411	226	116	995	101	10	265	206
vcscvs	529	212	71	0	49	0	0	0	0
vcsgeneric			34	0	641	111	2	169	102
xml				0					198
web	143	215	0	0	284	23	0	485	770
SUM	3955	5077	889	910	6985	1109	369	6418	6743
The Max Inc	547	686	226	320	995	111	71	1854	1236
Package name	core	form	vcscore	javacvs	vcscore	vcsgen-eric	core	core	schema 2beans

Table A.3 and Table A.4 were used in analysing the third research question. They were created by taking the columns that indicates to the maximum increase in the number of the attributes and the methods for each package across the nine versions.

Table A.3 The max. increase in the NOA for each package (Velocity)

Version No.\ Package name	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
ant		3	0	2	5	0	0	0	4
apisupport		1	0	0	3	1	0	5	
applet	2	0	0	0	0	0	0	0	0
autoupdate	1	7	1	2	10	1	1	6	5
beans	1	3	0	0	5	0	0	0	8
classfile				0		0	1	2	1
clazz	0	5	0	0	3	0	0	1	0
core	26	11	2	2	23	5	3	57	26
debuggercore	5	3	1	0	18	10	1	5	2
debuggerjpda	3	11	0	0	11	0	0	2	4
debuggertools	3	11	0		2	0	0		
diff				7					3
editor	9	7	1	1	12	3	0	12	19
extbrowser			0	0	4	1	0	9	0
form	8	49	0	0	51	0	0	15	8
html	0	0	0	0	1	0	0	0	0
httpserver	1	3	0	0	3	0	0	0	1
i18n	10	5	1	0	4	0	4	3	1
icebrowser	0	3	0						
image	2	1	0	0	0	0	0	0	2
j2eeserver				0		0	0	2	4
jarpackager	1	7	0	0	7	0	0	2	7
java	18	12	1	0	16	1	2	9	34
javacvs			1	3	11	1	1	7	13
javadoc	7	5	0	1	2	1	0	7	2
jndi			7	0	1	1	0	7	0
nbbuild	0	1	0	0	2	3	0	6	13
objectbrowser	0	0	0						
openide	7	26	1	3	26	4	1	15	9
openidex	0	0	0	0	2	0	0	0	0
projects	3	27	0	0	3	1	3	1	9
properties	2	6	0	0	1	0	0	0	0
rmi			3	0	10	0	0	1	
schema2beans				0					16

scripting		1	0	0	3	0	0	2	
text	1	1	0	0	1	0	0	0	0
tomcatint				0		0	0	10	2
ui				1					6
usersguide			0	0	0	0	0	2	0
utilities	9	13	0	0	3	1	0	1	1
vcscore		28	4	2	43	2	0	16	8
vcscvs	20	10	2	0	3	0	0	0	0
vcsgeneric			0	0	32	2	0	13	2
xml				0					6
web	14	2	0	0	4	1	0	20	10
SUM	153	262	25	24	325	39	17	238	226
The Max Inc	26	49	7	7	51	10	4	57	34
Package name	core	form	jndi	diff	form	debug-ercore	i18n	core	java

Table A.04 The max. increase in the NOM for each package (Velocity)

Version No.\ Package name	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6	Ver7	Ver8	Ver9
ant		6	0	3	9	1	0	2	2
apisupport		2	0	0	4	1	0	3	
applet	3	1	0	0	0	0	0	0	2
autoupdate	1	8	1	0	8	4	3	6	13
beans	5	6	0	1	5	0	0	1	4
classfile				0		0	2	2	0
clazz	3	4	0	0	10	0	0	1	0
core	17	20	3	1	48	7	6	150	22
debuggercore	9	6	1	0	20	1	3	5	13
debuggerjpda	3	12	0	0	9	0	0	2	2
debuggertools	6	11	0		15	1	0		
diff				6					1
editor	14	9	0	2	32	4	0	20	29
extbrowser			0	0	6	0	0	3	0
form	11	20	8	0	52	4	1	10	26
html	1	2	0	0	1	0	0	3	1
httpserver	4	7	0	0	3	0	0	3	1
i18n	4	10	0	0	5	0	1	3	1
icebrowser	0	3	0						
image	4	11	0	0	1	0	0	7	1
j2eeserver				0		1	0	7	17
jarpackager	15	9	1	0	7	2	1	1	8
java	23	17	0	0	17	2	1	7	20

javacvs			9	1	26	1	0	8	16
javadoc	8	13	1	2	4	1	0	4	2
jndi			0	0	2	5	1	4	0
nbbuild	0	1	0	0	4	6	0	8	9
objectbrowser	0	0	0						
openide	13	9	1	3	40	14	2	19	13
openidex	0	2	0	0	3	1	0	0	0
projects	3	18	0	0	8	2	4	2	10
properties	5	4	0	0	4	1	0	6	1
rmi			4	0	13	1	2	1	
schema2beans				0					30
scripting		16	0	0	8	0	0	2	
text	3	2	0	0	1	0	0	0	0
tomcatint				0		0	0	36	6
ui				1					16
usersguide			0	0	0	0	0	1	0
utilities	9	12	0	0	5	2	0	1	3
vcscore		84	11	2	70	9	2	18	12
vcscvs	36	3	4	0	2	0	0	0	0
vcsgeneric			2	0	38	2	0	6	5
xml				0					76
web	28	7	0	0	9	0	0	19	16
SUM	228	335	46	22	489	73	29	371	378
The Max Inc	36	84	11	6	70	14	6	150	76
Package name	vcscvs	vcscore	vcscore	diff	vcscore	openide	core	core	xml

APPENDIX B: DATA USED IN ANALYSING 80/20 RULE IN THE COUPLING METRICS

Tables in this appendix show the percentage in each of the coupling metrics over the versions of the the system for 20% of the classes on a package basis. The value is bolded if 80% is found (An 80/20 rule applies *if* at least 80% of the coupling is incorporated in that top 20%).

Table B.1 shows the data for the Jasmin system.

Table B. 1 The percentage of coupling metrics in 20% of the classes (Jasmin)

Package	Version	RFC	MPC	EXT	PACK	FOUT	F-IN
Jas	V1	45.58	59.94	54.48	44.29	76.88	**90.15**
	V2	46.31	61.07	54.8	47.04	77.92	**89.35**
	V3	49.63	62.09	56.76	48	72.1	**83.41**
	V4	50.99	62.56	58.26	48	74.61	**84.78**
	V5	50.99	62.56	58.26	48	72	**84.78**
jasmin	V1	60.07	70.58	63.48	49.09	58.36	**90.71**
	V2	68.13	77.42	70.91	50.43	61.16	**92.9**
	V3	73.75	**81.39**	73	46.67	66.6	**94.19**
	V4	71.6	78.66	73.88	46	62.68	**92.9**
	V5	71.67	78.87	73.97	46	62.68	**93.33**

Table B.2 shows the same data for the SmallSQL system.

Table B.02 The percentage of coupling metrics in 20% of the classes (SmallSQL)

Package	Version	RFC	MPC	EXT	PACK	FOUT	F-IN
database	V1	62.1	72.55	65.75	73	74.79	**88.54**
	V2	62.25	72.74	65.71	73.79	75.24	**90.33**
	V3	63.48	73.97	66.95	75.19	75.53	**91.18**
	V4	63.39	73.84	66.74	75.66	75.34	**91.15**
	V5	63.36	73.44	66.89	74.95	75.05	**91.23**
	V6	64.4	74.08	67.13	75.27	74.68	**92.44**
	V7	64.61	74.31	67.38	75	74.35	**92.19**
	V8	64.51	74.13	67.12	73.04	74.01	**92.11**
	V9	64.12	73.45	66.48	73.28	73.9	**92**

Table B.3 shows the data for the DjVu system.

Table B.03 The percentage of coupling metrics in 20% of the classes (DjVu)

Package	Version	RFC	MPC	EXT	PACK	FOUT	F-IN
djvu	V1	48.07	65.26	55.87	41.13	48.28	54.54
	V2	48.05	65.03	55.53	40.37	47.73	54.72
	V3	50.76	67.02	58.82	46.55	49.31	51.9
	V4	50.76	67.08	58.82	46.55	49.43	52.17
	V5	51.21	67.08	58.82	46.55	49.43	53.12
	V6	51.21	67.08	58.82	46.55	49.43	52.12
	V7	51.26	67.89	58.73	44.67	48.91	55.54
	V8	51.16	67.79	58.66	44.67	48.32	55.3
djvu.anno	V1	57.09	60.68	57.5	54.29	76.36	52.5
	V2	57.09	60.68	57.5	54.29	76.36	52.5
	V3	47.65	59.12	54.53	46	78.29	53.33
	V4	47.65	59.12	54.53	46	78.29	53.33
	V5	55.27	64.46	60.6	38.57	*80.23*	56.43
	V6	55.27	64.46	60.6	38.57	*80.23*	56.43
	V7	55.38	64.68	60.79	38.57	*80.46*	56.43
	V8	56.2	65.32	62.11	38.57	*80.68*	56.43
Toolbar	V1	41.6	46.7	41.32	31.15	45.12	*83.81*
	V2	37.68	42.38	37.2	27.87	42.14	*81.9*
	V3	43.41	48.08	42.69	30	45.52	77.65
	V4	43.41	48.08	42.69	30	45.52	77.65
	V5	43.41	48.08	42.69	30	45.52	75.43
	V6	43.13	48.81	43.07	30	45.19	76.76
	V7	43.13	48.81	43.07	29.51	44.85	76.76
	V8	43.13	48.81	43.1	29.51	44.85	76.76

Table B.4 shows the trends for the pBeans system.

Table B.4 The percentage of coupling metrics in 20% of the classes (pBeans)

Package	Version	RFC	MPC	EXT	PACK	FOUT	F-IN
Pbeans	V1	78.12	**90.98**	**85.73**	**85**	**88.32**	73.22
	V2	77.9	**90.79**	**85.67**	**85**	**88.32**	73.45
	V3	78.05	**91.89**	**86.56**	**81.05**	**87.29**	72.12
	V4	78.05	**91.89**	**86.56**	**81.05**	**87.29**	72.12
	V5	78.4	**92.23**	**87.1**	**81.05**	**87.29**	72.12
	V6	78.4	**92.23**	**87.1**	**81.05**	**87.29**	72.12
	V7	78.95	**94.02**	**89.23**	**81.82**	**90.6**	67.48
	V8	75.94	**89.5**	**84.67**	**83.9**	**85.69**	69.96
	V9	75.94	**89.5**	**84.67**	**83.9**	**85.69**	69.96
	V10	76.11	**89.54**	**84.74**	**83.9**	**85.69**	70.2
Data	V1	64.56	79.57	77.5	36.67	73.53	**82.86**
	V2	63.86	79.3	77.11	38.71	71.23	**87.5**
	V3	63.18	77.1	73.17	37.78	68.3	**92.89**
	V4	63.75	77.43	72.48	37.78	68.63	**92.89**
	V5	63.75	77.43	72.48	37.78	68.63	**92.89**
	V6	63.15	75.85	71.31	39.51	67.96	**94.78**
	V7	62.65	75.98	70.97	39.51	66.49	**99.53**
	V8	70.79	**89.64**	**85.08**	59.13	**83.76**	**81.14**
	V9	69.9	**88.86**	**84.38**	59.13	**80.22**	79.63
	V10	69.63	**88.26**	**83.85**	59.13	79.34	79.63

APPENDIX C: DATA USED IN ANALYSING THE FIN AND FOUT RELATIONSHIP

The tables in this appendix provide detail on name of classes, NOM, LOC, RFC, MPC, EXT, PACK, FOUT and FIN for the classes with 80% of the FIN in each of the large package across the released versions for the six systems. We used these tables to produce the figures in analysing RQ2 in Chapter 7.

Table C.01 Metrics values for Jas package (Jasmin)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	FIN
CatchEntry	3	19	7	6	4	1	0	4
ClassEnv	15	102	48	55	33	4	8	4
CodeAttr	12	98	43	41	31	2	5	9
GenericAttr	5	16	11	6	6	1	0	8
Insn	8	56	21	16	13	2	0	6
Label	6	24	10	5	4	1	1	18
LocalVarEntry	4	23	9	6	5	1	0	4
LongCP	3	10	6	3	3	1	1	1
Method	3	28	10	7	7	1	0	3
Var	3	23	8	5	5	1	0	2
CatchEntry	4	30	8	6	4	1	0	4
ClassEnv	20	134	63	65	43	4	8	4
CodeAttr	13	109	47	44	34	2	5	9
GenericAttr	5	16	11	6	6	1	0	8
Insn	10	68	24	18	14	2	0	6
Label	6	24	10	5	4	1	1	12
LabelOrOffset	5	15	7	2	2	2	1	15
LocalVarEntry	5	34	10	7	5	1	0	4
StackMapFrame	6	28	15	17	9	3	3	8
VerificationTypeInfo	4	45	14	10	10	1	2	8
Annotation	11	66	30	33	19	3	3	15
ClassEnv	22	181	76	76	54	4	11	4
CodeAttr	14	120	51	47	37	2	5	9
GenericAttr	6	24	17	12	11	1	2	17
InnerClass	4	39	10	7	6	1	0	4
Insn	11	77	27	22	16	2	0	6

Label	6	24	10	5	4	1	1	12
LabelOrOffset	5	15	7	2	2	2	1	13
LocalVarEntry	5	34	10	7	5	1	0	8
VerificationTypeInfo	4	50	16	12	12	1	2	4
Annotation	11	66	30	33	19	3	3	15
CodeAttr	14	120	51	47	37	2	5	9
GenericAttr	6	24	17	12	11	1	2	17
InnerClass	4	39	10	7	6	1	0	4
Insn	11	77	27	22	16	2	0	6
Label	7	26	11	6	4	1	1	13
LabelOrOffset	5	15	7	2	2	2	1	13
LocalVarEntry	5	34	10	7	5	1	0	8
VerificationTypeInfo	6	62	21	16	15	1	2	8
VerifyFrame	11	110	32	34	21	3	10	9
Annotation	11	66	30	33	19	3	3	15
CodeAttr	14	120	51	47	37	2	5	9
GenericAttr	6	24	17	12	11	1	2	17
InnerClass	4	39	10	7	6	1	0	4
Insn	11	77	27	22	16	2	1	6
Label	7	26	11	6	4	1	1	13
LabelOrOffset	5	15	7	2	2	2	1	13
LocalVarEntry	5	34	10	7	5	1	0	8
VerificationTypeInfo	6	62	21	16	15	1	2	8
VerifyFrame	11	110	32	34	21	3	10	9

Table C.02 Metrics values for Jasmin package (Jasmin)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	FIN
InsnInfo	3	12	6	4	3	2	1	17
ScannerUtils	7	63	26	29	19	0	8	8
InsnInfo	3	12	6	4	3	2	1	19
ScannerUtils	7	68	27	31	20	0	10	9
InsnInfo	3	12	6	4	3	2	1	19
ScannerUtils	7	68	26	31	19	0	9	9
InsnInfo	3	12	6	4	3	2	1	19
ScannerUtils	7	68	26	31	19	0	9	9
InsnInfo	3	12	6	4	3	2	1	21
ScannerUtils	7	68	26	31	19	0	9	9

Table C. 3 Metrics values for Database package (SmallSQL)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
CommandSelect	35	254	101	119	66	1	19	8
Database	19	192	85	145	66	3	25	11
DataSource	15	18	15	0	0	0	1	8
DateTime	27	424	60	62	33	4	17	41
Expression	33	80	43	13	10	0	12	81
ExpressionArithmetic	35	424	104	196	69	1	70	9
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	11
ExpressionValue	35	292	101	104	66	2	54	30
IndexDescription	16	73	49	41	33	3	16	12
IndexNode	32	176	64	55	32	1	7	29
Money	17	52	29	13	12	1	4	27
MutableNumeric	37	334	78	76	41	1	14	29
SQLToken	3	14	4	1	1	0	0	34
SQLTokenizer	4	156	17	14	13	2	61	43
SSResultSetMetaData	28	111	49	46	21	1	52	8
StoreImpl	70	650	217	408	147	2	65	22
StorePage	5	25	8	3	3	2	2	12
Strings	5	17	8	3	3	0	2	14
Table	19	234	81	94	62	6	18	16
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	9	50	28	25	19	2	8	17
TableViewResult	10	32	16	7	6	1	8	11
Utils	25	211	54	37	29	1	12	245
Column	24	69	36	12	12	2	10	16
CommandSelect	35	254	101	119	66	1	19	8
Database	21	211	97	161	76	3	27	14
DataSource	15	18	15	0	0	0	1	8
DateTime	27	424	60	62	33	4	17	41
Expression	33	80	43	13	10	0	12	81
ExpressionArithmetic	35	424	104	196	69	1	70	9
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	11
ExpressionValue	35	292	101	104	66	2	54	30
IndexDescription	16	73	49	41	33	3	16	12
IndexNode	32	176	64	55	32	1	7	29

Money	17	52	29	13	12	1	4	27
MutableNumeric	37	334	78	76	41	1	14	29
SQLToken	3	14	4	1	1	0	0	34
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	111	49	46	21	1	52	8
StoreImpl	70	653	217	410	147	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	5	17	8	3	3	0	2	14
Table	19	229	78	91	59	6	18	16
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	19
TableViewResult	10	32	16	7	6	1	8	11
Utils	25	211	54	37	29	1	12	358
CommandSelect	35	254	101	119	66	1	19	8
Database	21	211	98	161	77	3	27	14
DataSource	15	18	15	0	0	0	1	8
DateTime	26	429	59	61	33	4	17	44
Expression	33	80	43	13	10	0	12	84
ExpressionArithmetic	35	424	104	196	69	1	70	9
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	11
ExpressionValue	35	294	101	104	66	2	55	30
IndexDescription	17	76	52	43	35	3	16	13
IndexNode	32	176	64	55	32	1	7	29
Money	17	52	29	13	12	1	4	27
MutableNumeric	37	334	78	76	41	1	14	29
SQLToken	3	14	4	1	1	0	0	34
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	111	49	46	21	1	52	8
StoreImpl	70	654	218	411	148	2	67	22
StorePage	5	25	8	3	3	2	2	12
Strings	5	17	8	3	3	0	2	14
Table	19	232	82	93	63	6	19	16
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	19
TableViewResult	10	32	16	7	6	1	8	11
Utils	27	223	56	37	29	1	16	361
Column	24	69	36	12	12	2	10	16

CommandSelect	35	254	101	119	66	1	19	8
Database	21	217	102	167	81	3	28	14
DataSource	15	18	15	0	0	0	1	8
DateTime	26	429	59	61	33	4	17	44
Expression	33	80	43	13	10	0	12	84
ExpressionArithmetic	35	424	104	196	69	1	70	9
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	11
ExpressionValue	35	294	101	104	66	2	55	32
IndexDescription	17	76	52	43	35	3	16	13
IndexNode	32	176	64	55	32	1	7	29
Money	17	52	30	14	13	1	5	27
MutableNumeric	38	324	81	76	43	1	13	28
SQLToken	3	14	4	1	1	0	0	34
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	114	49	46	21	1	57	8
StoreImpl	70	656	216	410	146	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	5	17	8	3	3	0	2	14
Table	19	232	82	93	63	6	19	16
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	19
TableViewResult	10	32	16	7	6	1	8	11
Utils	27	223	56	37	29	1	16	366
Column	24	69	36	12	12	2	10	18
CommandSelect	36	256	102	119	66	1	19	9
Database	21	217	102	167	81	3	28	14
DataSource	15	18	15	0	0	0	1	8
DateTime	26	429	59	61	33	4	17	44
Expression	33	80	43	13	10	0	12	84
ExpressionArithmetic	35	426	104	198	69	1	70	9
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	11
ExpressionValue	35	294	101	104	66	2	55	34
IndexDescription	17	76	52	43	35	3	16	13
IndexNode	32	176	64	55	32	1	7	29
Money	17	52	30	14	13	1	5	27
MutableNumeric	38	324	81	76	43	1	13	28
SQLToken	3	14	4	1	1	0	0	34

SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	114	49	46	21	1	57	8
StoreImpl	70	661	216	410	146	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	6	20	9	4	3	0	2	15
Table	19	232	82	93	63	6	19	16
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	19
TableViewResult	10	32	16	7	6	1	8	11
Utils	27	223	56	37	29	1	16	370
Column	24	69	36	12	12	2	10	18
CommandSelect	36	259	102	119	66	1	18	9
Database	21	222	102	171	81	3	28	14
DataSource	15	18	15	0	0	0	1	8
DateTime	27	435	60	61	33	4	18	40
Expression	33	80	43	13	10	0	12	85
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	12
ExpressionValue	35	302	105	109	70	2	55	34
IndexDescription	17	76	52	43	35	3	16	14
IndexNode	32	176	64	55	32	1	7	29
LongTreeList	21	264	41	43	20	1	4	7
Money	17	52	30	14	13	1	5	33
MutableNumeric	38	324	81	76	43	1	13	30
SQLToken	3	14	4	1	1	0	0	35
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	114	49	46	21	1	57	9
StoreImpl	70	661	216	410	146	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	6	20	9	4	3	0	2	16
Table	19	238	87	98	68	6	21	19
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	20
TableViewResult	10	32	16	7	6	1	8	11
Utils	28	234	57	37	29	1	16	375
Column	23	68	35	12	12	2	10	18
CommandSelect	36	259	102	119	66	1	18	9
Database	21	222	102	171	81	3	28	14

DataSource	15	18	15	0	0	0	1	8
DateTime	28	437	62	62	34	4	18	43
Expression	33	80	43	13	10	0	12	89
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	13
ExpressionValue	35	302	107	111	72	2	55	35
IndexDescription	19	100	60	53	41	4	18	14
IndexNode	33	178	67	57	34	1	7	29
LongTreeList	21	264	41	43	20	1	4	7
Money	17	52	30	14	13	1	5	33
MutableNumeric	42	332	85	76	43	1	13	30
SQLToken	3	14	4	1	1	0	0	35
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	114	49	46	21	1	58	9
StoreImpl	70	661	220	415	150	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	6	20	9	4	3	0	2	16
Table	19	240	88	99	69	6	21	19
TableStorePage	4	20	8	4	4	1	2	32
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	20
TableViewResult	10	32	16	7	6	1	8	11
Utils	29	236	58	38	29	1	16	398
Column	23	68	35	12	12	2	10	20
Columns	7	29	14	9	7	0	3	9
CommandSelect	36	264	103	121	67	1	18	9
Database	24	243	113	189	89	3	29	16
DataSource	15	18	15	0	0	0	1	8
DateTime	27	445	64	62	37	4	19	43
Expression	34	87	45	14	11	1	13	91
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	14
ExpressionValue	35	304	109	115	74	2	55	35
IndexDescription	19	100	60	53	41	4	18	15
IndexNode	33	178	67	57	34	1	7	29
Money	17	52	30	14	13	1	5	33
MutableNumeric	42	336	85	76	43	1	13	30
SQLToken	3	14	4	1	1	0	0	38
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	116	49	46	21	1	58	9

StoreImpl	70	662	220	415	150	2	66	22
StorePage	5	25	8	3	3	2	2	12
Strings	6	20	9	4	3	0	2	17
Table	20	266	89	102	69	6	22	23
TableStorePage	4	20	8	4	4	1	2	36
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	20
TableViewResult	10	32	16	7	6	1	8	11
Utils	33	253	67	44	34	1	17	409
Column	23	68	35	12	12	2	10	21
Columns	7	33	16	9	9	0	3	9
CommandSelect	36	264	103	121	67	1	18	9
Database	24	243	113	189	89	3	29	16
DataSource	15	18	15	0	0	0	1	8
DateTime	27	464	64	62	37	4	19	43
Expression	34	87	45	14	11	1	13	91
ExpressionName	31	83	60	30	29	0	4	22
Expressions	15	59	20	13	5	0	2	14
ExpressionValue	35	304	109	115	74	2	55	35
IndexDescription	19	100	60	53	41	4	18	15
IndexNode	33	178	66	56	33	1	7	28
Money	17	52	30	14	13	1	5	33
MutableNumeric	42	336	85	76	43	1	13	30
SQLToken	3	14	4	1	1	0	0	38
SQLTokenizer	4	156	17	14	13	2	61	91
SSResultSetMetaData	28	116	49	46	21	1	58	9
StoreImpl	72	675	222	419	150	2	67	30
StorePage	5	25	8	3	3	2	2	15
Strings	6	20	9	4	3	0	2	17
Table	20	266	89	102	69	6	22	23
TableStorePage	4	20	8	4	4	1	2	36
TableStorePageInsert	3	11	4	1	1	1	1	8
TableView	10	51	29	25	19	2	8	20
TableViewResult	10	32	16	7	6	1	8	11
Utils	34	257	69	46	35	2	19	416

Table C. 4 Metrics values for Djvu package (DjVu)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
DataPool	18	84	46	36	28	2	7	29
DjVuOptions	57	115	61	6	4	0	2	26
GBitmap	38	476	87	99	49	1	8	22
GPixel	21	56	32	32	11	0	3	29
GPixelReference	14	35	16	3	2	2	0	27
GPixmap	39	498	102	250	63	2	11	18
GRect	14	82	20	15	6	0	4	59
JB2Shape	7	25	15	8	8	2	5	20
DataPool	20	95	52	40	32	2	8	30
DjVuOptions	59	124	65	8	6	1	4	28
GBitmap	38	476	87	99	49	1	8	22
GPixel	21	56	32	32	11	0	3	29
GPixelReference	14	35	16	3	2	2	0	27
GPixmap	39	498	102	250	63	2	11	18
GRect	14	82	20	15	6	0	4	59
JB2Shape	7	25	15	8	8	2	5	20
DataPool	23	109	63	49	40	3	8	35
DjVuObject	9	59	20	14	11	4	9	26
DjVuOptions	57	116	61	6	4	1	2	26
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29
GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	8	25
GRect	15	82	21	15	6	0	4	78
DataPool	23	109	63	49	40	3	8	35
DjVuObject	9	59	20	14	11	4	9	26
DjVuOptions	57	116	61	6	4	1	2	27
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29
GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	8	27
GRect	15	82	21	15	6	0	4	78
DataPool	23	109	63	49	40	3	8	35
DjVuObject	9	59	20	14	11	4	9	26
DjVuOptions	61	124	65	6	4	1	2	29
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29

GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	8	27
GRect	15	82	21	15	6	0	4	81
DataPool	23	109	63	49	40	3	8	35
DjVuObject	9	59	20	14	11	4	9	28
DjVuOptions	61	124	65	6	4	1	2	29
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29
GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	8	27
GRect	15	82	21	15	6	0	4	81
CachedInputStream	28	137	59	44	31	3	14	68
DjVuObject	12	65	26	17	14	4	9	29
DjVuOptions	61	125	65	6	4	2	2	54
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29
GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	7	27
GRect	15	82	21	15	6	0	4	81
CachedInputStream	28	137	59	44	31	3	14	61
DjVuObject	12	65	26	17	14	4	9	29
DjVuOptions	61	125	65	6	4	2	2	54
GBitmap	28	242	69	72	41	1	9	26
GPixel	21	56	35	32	14	0	3	29
GPixelReference	17	79	34	25	17	2	2	28
GPixmap	16	413	81	209	65	0	7	27
GRect	15	82	21	15	6	0	4	81

Table C.05 Metrics values for Anno package (DjVu)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
GMapOval	6	14	16	10	10	2	4	3
GMapPoly	32	207	69	78	37	2	5	3
GMapOval	6	14	16	10	10	2	4	3
GMapPoly	32	207	69	78	37	2	5	3
Poly	32	207	69	78	37	2	5	3
Rect	47	196	65	32	18	2	4	4
Poly	32	207	69	78	37	2	5	3
Rect	47	196	65	32	18	2	4	4

Poly	32	220	69	77	37	2	5	4
Rect	59	221	80	41	21	2	4	11
Poly	32	220	69	77	37	2	5	4
Rect	59	221	80	41	21	2	4	11
Poly	32	220	69	77	37	2	5	4
Rect	59	221	80	41	21	2	4	11
Poly	32	220	69	77	37	2	5	4
Rect	59	221	80	41	21	2	4	11

Table C.6 Metrics values for Toolbar package (DjVu)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
ListenerSupport	6	46	18	13	12	7	5	2
ToggleButton	45	249	125	121	80	6	23	16
ListenerSupport	6	46	18	13	12	7	5	2
ToggleButton	45	249	125	121	80	6	23	16
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	249	125	121	80	6	23	16
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	249	125	121	80	6	23	16
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	249	125	121	80	6	23	16
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	249	125	121	80	6	23	18
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	250	125	121	80	7	23	18
ComboBox	30	168	101	95	71	4	19	13
ToggleButton	45	250	125	121	80	7	23	18

Table C.07 Metrics values for pBeans package (pBeans)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
Persistent	0	1	0	0	0	0	0	25
StoreException	3	4	3	0	0	0	0	52
StoreInfo	6	9	6	0	0	2	0	9
Persistent	0	1	0	0	0	0	0	25
StoreException	3	4	3	0	0	0	0	53
StoreInfo	6	9	6	0	0	2	0	9
Persistent	0	1	0	0	0	0	0	30
PersistentID	6	13	10	4	4	0	3	11
StoreException	3	4	3	0	0	0	0	56
Persistent	0	1	0	0	0	0	0	30
PersistentID	6	13	10	4	4	0	3	11
StoreException	3	4	3	0	0	0	0	56
Persistent	0	1	0	0	0	0	0	30
PersistentID	6	13	10	4	4	0	3	11
StoreException	3	4	3	0	0	0	0	56
Persistent	0	1	0	0	0	0	0	30
PersistentID	6	13	10	4	4	0	3	11
StoreException	3	4	3	0	0	0	0	56
Persistent	0	1	0	0	0	0	0	34
PersistentID	6	13	10	4	4	0	3	10
StoreException	3	4	3	0	0	0	0	57
StoreInfo	7	10	7	0	0	2	0	9
GlobalPersistentID	8	25	18	10	10	0	3	18
PersistentID	6	13	10	4	4	0	3	17
Store	85	690	293	433	208	10	65	30
StoreException	3	4	3	0	0	0	0	93
StoreInfo	15	18	15	0	0	2	0	14
GlobalPersistentID	8	25	18	10	10	0	3	18
PersistentID	6	13	10	4	4	0	3	17
Store	85	690	293	433	208	10	65	30
StoreException	3	4	3	0	0	0	0	93
StoreInfo	15	18	15	0	0	2	0	14
GlobalPersistentID	8	25	18	10	10	0	3	18
PersistentID	6	13	10	4	4	0	3	17
Store	88	696	298	436	210	10	65	30
StoreException	3	4	3	0	0	0	0	95
StoreInfo	15	18	15	0	0	2	0	14

Table C.08 Metrics values for Data package (pBeans)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
AbstractDatabase	38	330	165	193	127	4	35	4
FieldDescriptor	11	23	11	0	0	3	0	15
IndexDescriptor	9	25	16	8	7	1	1	10
ResultsIterator	1	2	1	0	0	0	0	3
IndexDescriptor	9	25	16	8	7	1	1	10
FieldDescriptor	11	23	11	0	0	3	0	15
FieldDescriptor	11	23	11	0	0	3	0	24
IndexDescriptor	9	25	16	8	7	1	1	14
ResultsIterator	1	2	1	0	0	0	0	3
FieldDescriptor	11	23	11	0	0	3	0	24
IndexDescriptor	9	25	16	8	7	1	1	14
ResultsIterator	1	2	1	0	0	0	0	3
FieldDescriptor	11	23	11	0	0	3	0	24
IndexDescriptor	9	25	16	8	7	1	1	14
ResultsIterator	1	2	1	0	0	0	0	3
FieldDescriptor	11	23	11	0	0	3	0	24
IndexDescriptor	9	25	16	8	7	1	1	15
ResultsIterator	1	2	1	0	0	0	0	3
FieldDescriptor	11	23	11	0	0	3	0	24
IndexDescriptor	9	25	16	8	7	1	1	15
ResultsIterator	1	2	1	0	0	0	0	3
Database	16	19	16	0	0	2	0	10
FieldDescriptor	14	28	15	1	1	0	0	48
IndexDescriptor	10	28	17	8	7	1	1	26
Database	16	19	16	0	0	2	0	10
FieldDescriptor	14	28	15	1	1	0	0	48
IndexDescriptor	10	28	17	8	7	1	1	26
Database	16	19	16	0	0	2	0	10
FieldDescriptor	14	28	15	1	1	0	0	48
IndexDescriptor	10	28	17	8	7	1	1	26

Table C.9 Metrics values for Fastagi package (Aserisk)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
AGIException	3	4	3	0	0	0	0	32
AGIReader	2	4	2	0	0	1	0	3
AGIRequest	18	20	18	0	0	1	0	16
AGIScript	1	2	1	0	0	0	0	7
AGIWriter	1	3	1	0	0	1	0	3
AGIException	3	4	3	0	0	0	0	32
AGIReader	2	4	2	0	0	1	0	3
AGIRequest	19	21	19	0	0	1	0	18
AGIScript	1	2	1	0	0	0	0	7
AGIConnectionHandler	5	42	25	20	20	8	11	5
AGIException	3	4	3	0	0	0	0	74
AGIRequest	19	21	19	0	0	1	0	18
AGIScript	1	2	1	0	0	0	0	7
AGIConnectionHandler	5	42	25	20	20	8	11	5
AGIException	3	4	3	0	0	0	0	107
AGIRequest	23	26	23	0	0	2	0	17
AGIScript	1	2	1	0	0	0	0	8
AGIConnectionHandler	5	42	26	21	21	8	11	5
AGIException	3	4	3	0	0	0	0	119
AGIRequest	27	30	27	0	0	2	0	22
AGIScript	1	2	1	0	0	0	0	8
AgiChannel	52	55	52	0	0	2	0	2
AgiException	2	3	2	0	0	0	0	109
AgiScript	1	2	1	0	0	0	0	12
MappingStrategy	1	2	1	0	0	0	0	12
AgiChannel	54	57	54	0	0	2	0	2
AgiException	2	3	2	0	0	0	0	109
AgiScript	1	2	1	0	0	0	0	17
MappingStrategy	1	2	1	0	0	0	0	12

Table C.10 Metrics values for Manager package (Asterisk)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
AsteriskServer	8	35	13	5	5	1	1	14
Channel	28	85	49	21	21	2	2	44
ChannelStateEnum	3	11	5	2	2	3	1	8

DefaultManagerConnection	31	264	113	107	82	25	21	6
Queue	7	20	11	4	4	4	1	8
TimeoutException	1	2	1	0	0	0	0	9
AsteriskServer	8	35	13	5	5	1	1	14
Channel	28	85	49	21	21	2	2	44
ChannelStateEnum	3	11	5	2	2	3	1	8
DefaultManagerConnection	31	262	114	108	83	25	21	5
Queue	7	20	11	4	4	4	1	8
TimeoutException	1	2	1	0	0	0	0	9
AsteriskServer	8	35	13	5	5	1	1	14
Channel	25	97	46	24	21	4	5	49
ChannelStateEnum	3	11	5	2	2	3	1	8
Extension	9	33	21	12	12	2	2	8
Queue	8	27	21	13	13	4	3	12
TimeoutException	1	2	1	0	0	0	0	7
AsteriskServer	8	35	13	5	5	1	1	14
Channel	28	106	53	30	25	4	5	49
ChannelStateEnum	3	11	5	2	2	3	1	8
Extension	9	33	21	12	12	2	2	11
Queue	8	27	21	13	13	4	3	12
TimeoutException	1	2	1	0	0	0	0	8
AsteriskServer	8	35	13	5	5	1	1	14
Channel	28	106	53	30	25	4	5	49
DefaultManagerConnection	37	344	136	130	99	32	31	6
Extension	9	33	21	12	12	2	2	11
Queue	8	27	21	13	13	4	3	12
TimeoutException	1	2	1	0	0	0	0	8
EventTimeoutException	2	5	2	0	0	0	0	3
ManagerConnection	17	23	17	0	0	5	0	6
ManagerEventListener	1	4	1	0	0	2	0	6
ResponseEvents	2	6	2	0	0	3	0	4
SendActionCallback	1	3	1	0	0	1	0	4
TimeoutException	1	2	1	0	0	0	0	7
DefaultManagerConnection	33	76	58	37	25	5	2	3
ManagerConnection	20	26	20	0	0	5	0	6
ManagerEventListener	1	4	1	0	0	2	0	6
ResponseEvents	2	6	2	0	0	3	0	4
SendActionCallback	1	3	1	0	0	1	0	4
TimeoutException	1	2	1	0	0	0	0	7

Table C.011 Metrics values for Manager.event package (Asterisk)

Class Name	NOM	LOC	RFC	MPC	EXT	PACK	FOUT	F-IN
ConnectEvent	3	6	3	0	0	0	0	13
DisconnectEvent	1	2	1	0	0	0	0	19
HangupEvent	3	6	3	0	0	0	0	5
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	4	14	11	7	7	3	2	18
NewChannelEvent	1	2	1	0	0	0	0	9
NewExtenEvent	15	30	15	0	0	0	0	8
NewStateEvent	1	2	1	0	0	0	0	5
QueueEntryEvent	11	22	11	0	0	0	0	5
QueueMemberEvent	13	26	13	0	0	0	0	5
QueueParamsEvent	17	34	17	0	0	0	0	5
RenameEvent	7	14	7	0	0	0	0	5
ShutdownEvent	5	10	5	0	0	0	0	5
StatusCompleteEvent	1	2	1	0	0	0	0	8
StatusEvent	23	46	23	0	0	0	0	5
UnlinkEvent	1	2	1	0	0	0	0	5
ConnectEvent	3	6	3	0	0	0	0	13
DisconnectEvent	1	2	1	0	0	0	0	19
HangupEvent	3	6	3	0	0	0	0	5
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	4	14	11	7	7	3	2	18
NewChannelEvent	1	2	1	0	0	0	0	9
NewExtenEvent	15	30	15	0	0	0	0	8
NewStateEvent	1	2	1	0	0	0	0	5
QueueEntryEvent	11	22	11	0	0	0	0	5
QueueMemberEvent	13	26	13	0	0	0	0	5
QueueParamsEvent	17	34	17	0	0	0	0	5
RenameEvent	7	14	7	0	0	0	0	5
ShutdownEvent	5	10	5	0	0	0	0	5
StatusCompleteEvent	1	2	1	0	0	0	0	8
StatusEvent	23	46	23	0	0	0	0	5
UnlinkEvent	1	2	1	0	0	0	0	5
ConnectEvent	3	6	3	0	0	0	0	11
DisconnectEvent	1	2	1	0	0	0	0	14
HangupEvent	5	10	5	0	0	0	0	6
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	6	21	14	8	8	3	2	28

NewCallerIdEvent	12	36	16	4	4	0	1	11
NewChannelEvent	1	2	1	0	0	0	0	9
NewExtenEvent	15	30	15	0	0	0	0	8
NewStateEvent	1	2	1	0	0	0	0	5
QueueEntryEvent	13	26	13	0	0	0	0	5
QueueMemberEvent	17	34	17	0	0	0	0	5
QueueParamsEvent	19	38	19	0	0	0	0	5
RenameEvent	7	14	7	0	0	0	0	5
ResponseEvent	5	10	5	0	0	0	0	11
ShutdownEvent	5	10	5	0	0	0	0	13
StatusCompleteEvent	1	2	1	0	0	0	0	6
StatusEvent	23	46	23	0	0	0	0	5
UnlinkEvent	1	2	1	0	0	0	0	5
ConnectEvent	3	6	3	0	0	0	0	11
DisconnectEvent	1	2	1	0	0	0	0	14
HangupEvent	5	10	5	0	0	0	0	6
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	6	21	14	8	8	3	2	28
NewCallerIdEvent	12	36	16	4	4	0	1	11
NewChannelEvent	1	2	1	0	0	0	0	9
NewExtenEvent	15	30	15	0	0	0	0	8
NewStateEvent	1	2	1	0	0	0	0	5
QueueEntryEvent	13	26	13	0	0	0	0	5
QueueMemberEvent	17	34	17	0	0	0	0	5
QueueParamsEvent	19	38	19	0	0	0	0	5
RenameEvent	7	14	7	0	0	0	0	5
ResponseEvent	5	10	5	0	0	0	0	11
ShutdownEvent	5	10	5	0	0	0	0	13
StatusCompleteEvent	1	2	1	0	0	0	0	6
StatusEvent	23	46	23	0	0	0	0	5
UnlinkEvent	1	2	1	0	0	0	0	5
ConnectEvent	3	6	3	0	0	0	0	11
DisconnectEvent	1	2	1	0	0	0	0	14
HangupEvent	5	10	5	0	0	0	0	6
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	6	21	14	8	8	3	2	28
NewCallerIdEvent	12	36	16	4	4	0	1	11
NewChannelEvent	1	2	1	0	0	0	0	9
NewExtenEvent	15	30	15	0	0	0	0	8
NewStateEvent	1	2	1	0	0	0	0	5

QueueEntryEvent	13	26	13	0	0	0	0	5
QueueMemberEvent	17	34	17	0	0	0	0	5
QueueParamsEvent	19	38	19	0	0	0	0	5
RenameEvent	7	14	7	0	0	0	0	5
ResponseEvent	5	10	5	0	0	0	0	11
ShutdownEvent	5	10	5	0	0	0	0	13
StatusCompleteEvent	1	2	1	0	0	0	0	6
StatusEvent	23	46	23	0	0	0	0	5
UnlinkEvent	1	2	1	0	0	0	0	5
CdrEvent	43	89	44	6	1	3	1	3
ConnectEvent	4	8	4	0	0	0	0	3
DisconnectEvent	1	2	1	0	0	0	0	4
HangupEvent	5	10	5	0	0	0	0	3
LinkEvent	1	2	1	0	0	0	0	3
ManagerEvent	6	34	17	11	11	6	7	7
MeetMeMuteEvent	3	6	3	0	0	0	0	3
MeetMeTalkingEvent	3	6	3	0	0	0	0	3
NewCallerIdEvent	4	20	8	4	4	0	2	3
NewChannelEvent	1	2	1	0	0	0	0	3
NewExtenEvent	15	30	15	0	0	0	0	3
NewStateEvent	1	2	1	0	0	0	0	3
PeerlistCompleteEvent	3	6	3	0	0	0	0	3
QueueMemberEvent	17	34	17	0	0	0	0	3
QueueParamsEvent	19	38	19	0	0	0	0	3
RenameEvent	7	14	7	0	0	0	0	3
ResponseEvent	5	10	5	0	0	0	0	15
UnlinkEvent	1	2	1	0	0	0	0	3
CdrEvent	43	89	44	6	1	3	1	5
ConnectEvent	4	8	4	0	0	0	0	5
DisconnectEvent	1	2	1	0	0	0	0	6
HangupEvent	5	10	5	0	0	0	0	5
LinkEvent	1	2	1	0	0	0	0	5
ManagerEvent	8	38	19	11	11	6	7	7
MeetMeMuteEvent	3	6	3	0	0	0	0	5
MeetMeTalkingEvent	3	6	3	0	0	0	0	5
NewCallerIdEvent	4	20	8	4	4	0	2	5
NewChannelEvent	1	2	1	0	0	0	0	5
NewExtenEvent	15	30	15	0	0	0	0	5
NewStateEvent	1	2	1	0	0	0	0	5
OriginateResponseEvent	19	40	20	1	1	0	0	8

PeerlistCompleteEvent	3	6	3	0	0	0	0	5
QueueMemberEvent	21	42	22	2	1	0	0	5
QueueParamsEvent	19	38	19	0	0	0	0	5
RenameEvent	9	18	9	0	0	0	0	5
ResponseEvent	5	10	5	0	0	0	0	15
UnlinkEvent	1	2	1	0	0	0	0	5

APPENDIX D: THE REFACTORING DATA FOR ANTLR AND PDFBOX

Tables in this appendix show the data for the fifteen refactoring analysed in this Thesis for Antlr and PDFBox systems.

Table D.01 Refactorings for the Antlr system across 9 versions

Refactoring	Ver1	Ver2	Ver3	Ver4	Ver5
AddMethodParameter	2	7	0	1	0
EncapsulateDowncast	0	0	0	0	0
HideMethod	0	0	0	0	0
PullUpField	0	0	0	0	0
PullUpMethod	0	5	0	0	0
PushDownField	0	0	0	0	0
PushDownMethod	0	0	0	0	0
RemoveMethodParameter	0	1	0	0	0
RenameField	1	6	0	0	0
RenameMethod	1	6	1	0	0
EncapsulateField	0	0	0	0	0
MoveField	0	6	0	0	0
MoveMethod	0	6	0	0	0
ExtractSuperClass	0	0	0	0	0
ExtractSubClass	0	0	0	0	0

Table D.02 Refactorings for the PDFBox system across 9 versions

Refactoring	Ver1	Ver2	Ver3	Ver4	Ver5	Ver6
AddMethodParameter	0	3	0	0	0	7
EncapsulateDowncast	0	0	0	0	0	0
HideMethod	0	0	0	0	0	0
PullUpField	0	0	0	0	0	0
PullUpMethod	0	0	0	0	0	0
PushDownField	0	0	0	0	0	0
PushDownMethod	0	0	0	0	0	0
RemoveMethodParameter	0	0	6	0	0	1
RenameField	0	0	1	0	1	3
RenameMethod	1	0	2	4	0	7
EncapsulateField	0	0	0	0	0	1
MoveField	0	0	4	1	0	1
MoveMethod	1	0	8	0	0	7
ExtractSuperClass	0	0	1	0	0	0
ExtractSubClass	0	0	0	0	0	0