An Empirical Investigation of Inheritance Trends in Java OSS Evolution

A Thesis submitted for a degree of Doctor of Philosophy

by

Emal Nasseri

Department of Information Systems, Computing and Mathematics

Brunel University

United Kingdom

June 2009

Abstract

Inheritance is a salient feature of Object-Oriented (OO) paradigm which facilitates reuse and improves system comprehensibility in OO systems. The overall aim of inheritance is to model classes in a structured hierarchy where classes residing lower in the hierarchy (subclasses) can inherit the pre-existing functionality in the classes located higher up (superclasses) in the same line of hierarchy. Software maintenance and evolution are the process of making any modifications to a software system and upgrading its dynamic behaviour.

In this Thesis, we empirically investigate the trends of evolution of eight Java Open-Source Systems (OSS) from an inheritance perspective and model the propensity for changes of inheritance in those systems. The systems used as testbed in this Thesis represent a variety of application domains with varying sizes and amount of inheritance employed. There are several levels of granularity for inheritance evolution that may manifest a particular trend. This starts from the highest level (package) to lower class, method an attribute levels; and each level may show a different and yet an important pattern of evolution. We empirically investigate the changes of inheritance in the form of increases (additions) and decreases (deletions) in number of classes, methods and attributes. Our analysis also includes the movement of classes within and across an inheritance hierarchy which is another compelling facet of evolution of inheritance and may not be extrapolated through incremental changes only. It requires a finer-grained scrutiny of evolutionary traits of inheritance. In addition, the Thesis also explores the trends of class interaction within and across an inheritance hierarchy and problems embedded in a system that may lead to faults, from an inheritance perspective. The results demonstrate how inheritance is used in practice, problems associated with inheritance and how inheritance hierarchies evolve as opposed to that of a ‘system’. Overall results informed our understanding of the trends in changes of inheritance in the evolution of Java systems.

Declaration of Authorship

I would like to certify that the work in this Thesis, to the best of my knowledge, is original and the results of my own investigation, except as acknowledged. I would also like to certify that the work presented here has not been submitted, either in part or whole, for a degree at this or any other University.

Emal Nasseri

Date: 29 June 2009

Dedication

To my father, my mother, my brothers Ajmal and Romal, my sisters Nargis, Laila, Muzhgan, Khatera and my little sister Arzo, my fiancée Lubica and all my friends and family, for their love and support throughout these stressful years.

Acknowledgement

During the course of this research, there have been numerous people who have provided me with guidance, inspiration and support for completing this Thesis. I am indebted to my supervisor Dr. Steve Counsell for his continuous encouragement and support at all decisive stages of this research. His involvement proved vital to the completion of this Thesis. I would like to thank him for his useful advice on both academic and personal levels. I also owe thanks to Professor Martin Shepperd, my second supervisor, for his insightful and encouraging comments and suggestions throughout this research. I must also express my thanks to all my colleagues in the Department of Information Systems, Computing and Mathematics at Brunel University and those outside.

On the personal front, I would like to express my gratitude to my family and friends, in particular to my parents, for their continuous support and sincere prayers during the course of this research.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DIT	Depth of Inheritance Tree (Chidamber and Kemerer metric)
EXT	Number of External methods calls (JHawk metric)
HIER	Number of calls to methods within Hierarchy (JHawk metric)
LCOM	Lack of Cohesion Of the Methods (Chidamber and Kemerer metric)
MPC	Message Passing Coupling (Li and Henry metric)
MOOD	Metric for Object-Oriented design
NOA	Number of Attributes (Lorenz and Kidd metric)
NOC	Number of Children (Chidamber and Kemerer metric)
NOM	Number of Methods (Lorenz and Kidd metric)
OO	Object-Oriented
OSS	Open-Source Systems
RR	Reuse Ratio (Henderson-Sellers metric)
SE	Software Engineering
SR	Specialization Ratio (Henderson-Sellers metric)

List of Publications

<u>2009</u>

1. Nasseri, E. and Counsell, S. (2009) An Empirical study of Java System Evolution at the Method Level. *Accepted to appear in the Proceedings of the 7th International Conference on Software Engineering Research Management and Applications*. Haikou, Hainan Island, China.

2. Stopford, B., Counsell, S., Nasseri, E. (2009) Simulating software evolution with varying numbers of developers and validation using OSS. *Proceedings of the 20th Australian Software Engineering Conference.* Gold Coast, Australia. pp.13-22.

3. Nasseri, E., Counsell, S. and Shepperd, M. (2009) Class Movement and Re-location: an Empirical Study of Java Inheritance Evolution. *Accepted (subject to minor changes) to appear in the Journal of Systems and Software*.

4. Nasseri, E., and Counsell, S. (2009) An Evolutionary Study of Inheritance and Method Invocation in Java OSS. *Accepted (subject to minor changes) to appear in the Software Quality Journal*.

5. Nasseri, E., and Counsell, S. (2009) System evolution at the attribute level: an empirical study of three Java OSS and their refactorings. *Proceedings of the 31st International Conference on Information Technology Interface*. Cavtat, Dubrovnik, Croatia. pp.653-658.

<u>2008</u>

6. Nasseri, E., Counsell, S. and Shepperd, M. (2008) An Empirical Study of Evolution of inheritance in Java OSS. *Proceedings of the 19th Australian Software Engineering Conference*. Perth, Australia. pp.269-278.

7. Nasseri, E. and Counsell, S. (2008) Inheritance, ‘Warnings’ and potential Refactorings. *Proceedings of the 3rd International Conference on Software Engineering Advances*. Sliema, Malta. pp.132-39.

CHAPTER 1 Introduction

1.1 Introduction

The Object-Oriented (OO) software development technology was initially introduced in the early 1990's. Since then the OO paradigm has dominated mainstream software development (both in academia and industry) with languages such as C++ and Java. OO technology employs '*Classes*' together with '*Objects*' and their interdependencies to design and implement systems. OO introduced various underpinning approaches (i.e., inheritance, polymorphism, and encapsulation mechanisms) to software development which distinguish OO from traditional software development paradigm.

Inheritance is a cornerstone of OO paradigm. It is used to encapsulate a set of closely related functionality in a structured hierarchy where common functionality is added in one class (the superclass) and more specialized functionality of that class is added in other classes (its subclasses). The specialized classes inherit the common functionality from their superclass and add their own extra functionality. The primary concern of inheritance is to promote reusability in a system. The overriding merits of reusability in software development are to: (i) remove the burden of re-writing an existing segment(s); and (ii) to ease extensibility in a system. Furthermore, inheritance provides the facility for polymorphism.

In Java, a class can only inherit functionality from one other class, in C++ however, a class can inherit functionality from multiple classes in a system. To facilitate multiple inheritance Java introduced the notion of interfaces (see Section 2.2.3 for the difference between a class and interface).

In Software Engineering (SE), software evolution is a term used to refer to the development of a system and its continuous change; software maintenance is the process of making modifications to an existing system (Dvorak 1994). Software maintenance and evolution are two inter-related topics, and there is a growing concern about these two topics in the SE community. Empirical evidence exists to suggest that software maintenance accounts for a significant amount of software development cost (Lehman 1980b, Meyers 1988). The basic principle of software development is that systems should be designed in a way to accommodate easy *maintenance*.

From a maintainability perspective, *refactoring* seems to play a significant role in this sphere of software development activity (Fowler 1999). The key motivation behind refactoring, according to Fowler (Fowler 1999) is to improve system design and comprehensibility without making any modification to its external behaviour. In other words, refactoring can be used as an impediment to '*decay*' in code. Fowler (Fowler 1999) presents 72 types of refactorings with the motivation 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. Even one of Fowler's '*big*' refactorings 'Collapse Hierarchy' entails inheritance in its mechanism.

While the primary purpose of inheritance is to improve program comprehension and ease system maintenance, empirical evidence exists to suggest that use of inheritance can have the opposite effect (Harrison et al. 2000, Cartwright 1998, Cartwright and Shepperd 2000). The question; *does inheritance improve system comprehension and maintainability*? therefore remains unanswered. As a SE community, we know very little about the effect of inheritance and its limitations from a maintainability perspective. Previous studies suggest that inheritance should be used with care and only when necessary (Wood et al. 1999). In addition, we expect systems to evolve due to the changes of requirements and/or the environment in which they are operating. Previous studies have analyzed software systems from a maintenance and evolution perspective (Lehman 1974, Lehman et al, 1997, Kemerer and Slaughter 1999, Girba et al. 2005). What is not so obvious from these previous studies is how inheritance hierarchies in OO systems evolve in conjunction with, and as opposed to, that of system evolution. Moroever, inheritance is a form of coupling (Briand et al. 1999b). Anecdotal claims exist to suggest that coupling through inheritance is more favourable to that of non-inheritance coupling. We would therefore expect inheritance to be an alternative to non-inheritance coupling.

The purpose of the research in this Thesis is to thus investigate the changes of inheritance in the evolution of Java Open-Source Systems (OSS). In other words, we explore the trends that may exist in the changes of inheritance as systems evolve. Some interesting and insightful results relating to software evolution from an inheritance perspective emerged from the studies carried out.

The most striking result emerging from our investigation was that the vast majority of incremental changes (in terms of classes, methods and attributes) were made at '*shallow*' levels (levels one and two) of the class hierarchy, subverting the original aims of using inheritance. The overall results combined throughout this Thesis unveiled the pattern of incremental changes (in terms of classes, methods and attributes), class movement and relocation within and across inheritance hierarchy, method invocation within and across inheritance hierarchy and finally, the pattern of *'warnings'* (problems found in a class that may lead to faults) within an inheritance hierarchy.

1.2 Motivation

The motivation for conducting the empirical investigation in this Thesis stems from the following sources:

To build a body of knowledge on trends of inheritance feature which include: how changes (additions, deletion of classes, methods and attributes, movement of classes within inheritance hierarchies and method invocations and their evolution in a system) are made to inheritance hierarchies over time, a comparison of low level changes (i.e., methods and attributes) to that of applied low-level refactorings (i.e., method and attribute-related refactorings) and the impact on maintainability of inheritance, using warnings extracted by (FindBugs 2008), in Java OSS.

Empirical studies in SE to date have investigated inheritance and its implications on maintainability (Daly et al. 1996, Cartwright 1998, Harrison et al. 2000). However, from an evolutionary perspective, the pattern of change in inheritance is still unclear. Our knowledge and understanding of evolutionary forces of inheritance is almost negligible.

1.3 Objectives and Contribution

The primary objectives of this Thesis are:

1. To improve our understanding of inheritance in Java OSS. That is, to obtain a greater understanding of inheritance trends and how it is used in practice.
2. To investigate quantitatively how inheritance hierarchies evolve as opposed to that of system evolution as a whole. In particular, to conduct a thorough investigation of where in the inheritance hierarchy the majority of incremental changes are applied as a system evolves.
3. To investigate evolution of inheritance from a class movement and relocation perspective. In other words, to investigate how classes within an inheritance hierarchy are moved from one level to another as a system evolves.
4. To investigate inheritance from a class interaction perspective.

This Thesis makes a contribution in the realm of SE, in particular, from an evolutionary perspective, the results of which 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 1999) claimed that software evolution is scarcely researched and have expressed the need for further longitudinal empirical studies of software evolution. In particular, it is stressed that studies should take into consideration various levels of granularity when studying software evolution (Kemerer and Slaughter 1999). The research in this thesis is of importance for the following two reasons:

1. An appreciation of trends of inheritance can help predict future changes in the inheritance hierarchy in a system. In other words, the trends in inheritance can make change prediction, a challenging task, relatively easy.
2. The trends can help target future maintenance (i.e., refactoring) changes and take pre-emptive action to code decay in a system.

3. Since no empirical study to date has analyzed inheritance from an evolutionary perspective at various levels of granularity, we believe the methodological approaches adopted for data collection and analysis in this Thesis can help inform future empirical studies on inheritance and its evolution. That is, the approaches adopted in this Thesis can be used as a roadmap for further empirical studies of inheritance evolution.

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 inheritance in multiple versions of the studied systems.

1.4 Application Domains

The eight OSS used throughout this Thesis embody a variety of application domains. The rationale behind the selection of the systems from various application domains is to enable us to generalize the results of the studies conducted into a broader OO population. The subject OSS that we use in this Thesis are from the following application domains:

- A database system.
- Two game systems.
- A case-base reasoning system.
- A reporting engine.
- A library system.
- A Java application generator and
- A Java application server.

In addition, the set of OSS that we use have different sizes in terms of start and end number of classes, methods and attributes and contain various numbers of versions. The diversity of the application domains and the differences in the sizes of the systems allow conclusions to be drawn to a wider extent. Furthermore, the rationale behind using systems only in Java was that firstly, Java is newer than other OO languages. Secondly, Java is dominating the commercial software development community and finally, we would like to have an in-depth analysis of inheritance in one language rather than have a less detailed analysis of various languages. Further justification and criteria for sample selection is given in Chapter 2. The eight OSS are: HSQLDB, JColibri, JasperReports, EasyWay, SwingWT, JAG, JBoss and Tyrant. More details on each of these systems are given in Chapter 2.

1.5 Thesis Scope

The investigations described in this Thesis involve significant data collection from multiple versions of each system and various data analysis techniques from which to draw conclusions. The Thesis does not concern itself with other development aspects of the systems, such as the time intervals between each transition of versions, the number of developers working on each versions, the requirements (i.e., functional and/or non-functional) of the systems. In addition, formal system testing is also excluded from the scope of this Thesis.

1.6 Structure of the Thesis

This Thesis is organized into eight chapters:

Chapter 2 consists of two main sections. The first section presents a thorough survey of contemporary work in the areas of empirical SE, software metrics, inheritance, software maintenance and evolution, and finally software refactoring. The second section of Chapter 2 presents the methodology adopted for our research including, available research methods, our research design, including, sample selection and justification of the selected samples, data collection and statistical techniques employed.

Chapter 3 provides a description of an empirical study in which incremental changes of trends of inheritance from seven Java OSS described in Chapter 2 were investigated. This includes the trends of class changes in the same seven Java OSS and the changes at method and attribute level in a subset (three) of the systems. Results showed that approximately

96% of overall class changes were found at levels one and two of the class hierarchy. Only 4% of the same changes were found at levels three and beyond. In terms of methods and attributes, approximately 93% of method and 97% of attribute changes were made to classes at inheritance levels one and two. The remaining changes were made to classes at levels three and beyond.

Chapter 4 describes an empirical study in which changes of inheritance at method and attribute level are investigated in four OSS and the changes then compared with a set of low-level refactorings (i.e., method and attribute-related refactorings) applied to initial versions of the systems. Results revealed that analyzing a system at a lower-granularity (i.e., methods and attributes) can often show a different trend to that of a similar analysis at a higher-grain (i.e., classes and packages). Furthermore, our empirical results also indicated that analysis of a system at lower-granularity can often show trends that may go undetected when analyzing the system at a higher-granularity.

Chapter 5 presents an empirical investigation in which the trends in class movement and relocation within an inheritance hierarchy is explored. A sample of a number of versions from four Java OSS was selected and class movement and relocation examined. Results indicated that larger classes and tightly coupled classes were more frequently moved within the hierarchy then their respective smaller and loosely coupled classes. Furthermore, results also revealed that larger classes and tightly coupled classes were less cohesive which explained their movement within their respective class hierarchy.

Chapter 6 presents an empirical study of method invocation in four Java OSS. Result revealed that, due to the presence of a large number of classes at levels one and two of inheritance hierarchy, the majority of method calls were found at those two levels. It was also found that method invocation in a class tended to detract from class cohesion and class size was positively correlated to class coupling (through method calls).

Chapter 7 gives a description of an empirical study investigating the influence of inheritance on warnings (i.e., problems that may lead to potential faults) extracted by the FindBugs tool (FindBugs 2008), in three Java OSS. The investigation showed how inheritance hierarchies evolved and the propensity for generated warnings. The results also indicated how those warnings could be used to target refactorings in future releases of the systems.

Finally, Chapter 8 presents conclusions and contributions of the research presented in this Thesis with reflection on our original objectives and how they were achieved. In addition, the direction to possible future work is also given.

CHAPTER 2 Literature Survey and Methodology

2.1 Introduction

In the previous chapter we provided an introduction to the Thesis. The objective of this chapter is to provide a thorough survey of related work and the methodology used for our research. The chapter has three main sections, 2.2 Survey of Literature, 2.3 Methodology Adopted and 2.4 Summary. Each sub-section of the survey of literature presents the work of other researchers in one specific sphere. This includes empirical SE, software metrics, inheritance, software maintenance/evolution and software refactoring. We also demonstrate how previous work is related to our research and, equally, how our research is different and thus contributes to a body of knowledge in the field. The methodology section includes a description of the available research methods in SE and the design of the research presented in this Thesis. Finally, in Section 3.4 we present a summary of the chapter.

2.2 Survey of Literature

In this section, we present a thorough survey of studies related to our research. In Section 2.2.1 we discuss the related work in the realm of empirical SE. We then review the software metrics introduced in the literature and how they have been used in practice (Section 2.2.2). Section 2.2.3 provides a detailed analysis of published work on OO inheritance; In Section 2.2.4, we describe related research in software maintenance and evolution. Finally, we provide an analysis of published work on software refactoring (Section 2.2.5).

2.2.1 Empirical Software Engineering

The term Software Engineering (SE) refers to the discipline dealing with designing, developing, maintaining, testing and other aspects of complex software systems. The term *empirical* refers to observations and experiments. Empirical SE can be defined as 'the process of assessing the quality of software products, processes and projects in order to improve the current situation in SE'. Empirical SE is a diverse research area which has attracted the focus of numerous researchers investigating tools, methods, theories and other facets of SE. Kitchenham (Kitchenham. 2004) introduced the notion of systematic reviews into empirical SE, defined as: *"…a means of identifying, evaluating and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest."*

SE is considered as a young and immature discipline when compared to other engineering disciplines. The quality of empirical studies in SE has been subject to numerous criticisms. Researchers are continuously working on identifying the weaknesses of empirical research and consistently stress the need for improving empirical studies in SE (Fenton et al. 1994, Briand et al. 1999a, Seaman 1999). For example, Perry et al. (Perry et al. 2000) claimed that empirical studies in SE require improvement. They presented the strengths and weaknesses of empirical studies and proposed a number of steps to be taken in order to improve the current state of empirical SE research, including designing better studies, collecting and analysing constructive data and collaborating with other researchers in the field.

Briand et al. (Briand et al. 1999a) presented an overview of empirical studies of OO systems, methods and processes. They highlighted several key points to be considered in order to carry out successful empirical studies in SE. They firstly, encouraged close collaboration with the software industry. Secondly, they suggested improvements in the quality of empirical studies. Finally, they emphasised the need for replication of studies in SE. Kitchenham et al. (Kitchenham et al. 2002) introduced a set of guidelines for performing empirical studies in SE. They argued that the guidelines could be used to improve future empirical studies and could also help assess the quality of existing studies in the realm of empirical SE. They also claimed that the guidelines would be a good starting point for improvement of empirical studies in SE.

Seaman (Seaman 1999) presented a set of qualitative research methods for data collection and analysis in empirical SE research. In that study, it was illustrated how those qualitative methods could be used in practice. The author claimed that qualitative methods could also be used in conjunction with that of quantitative methods, and qualitative research methods could improve the quality and the amount of information contained in the dataset. O'Brien et al. (O'Brien et al. 2005) stressed the need for qualitative methods to be used in conjunction with quantitative methods in empirical SE research, to address the methodological shortfalls of experimental studies. Wood et al. (Wood et al. 1999) also showed the use of multi-method empirical research in SE. The multi-method approach was based on the combination of complementary empirical research methods, which were argued to solve the problems faced when conducting a single-shot empirical study. In addition, different research methods and desirable criteria for each method were explained.

The aforementioned studies have provided us with an understanding of empirical research in SE and revealed the strengths and weaknesses of studies in the field. As a result, we infer that SE provides ample opportunities for conducting empirical investigations. For example, the lack of empirical studies on inheritance, particularly from an evolutionary perspective, provided us with a motivation base to carry out empirical research concentrating on OO inheritance and its effective use in practice; more specifically the evolution of inheritance, changes of inheritance at various levels of granularity in the evolution of OO systems, problems associated with inheritance and opportunities that presented for refactoring. Our empirical investigation therefore builds a body of knowledge on inheritance and its evolution at various levels of granularity, bringing to light the strengths and weaknesses of inheritance from a maintenance perspective.

2.2.2 Software Metrics

In this Thesis, we present an empirical investigation of trends of inheritance in the evolution of Java OSS. Software metrics provide a quantitative basis for dimensions of a software project, process, or products and played a significant part in our research. That is, they enabled us to quantify inheritance in each version of the systems studied and to model the evolutionary behaviour of the systems from an inheritance perspective.

In the mid-1970's the phrase *software metrics* was first introduced by Tom Gilb (Gilb. 1976). Since then, the topic of software metrics has been a well researched in the SE research community. Using metrics, software practitioners, developers and researchers can understand, manage, plan and control characteristics of complex software systems. DeMarco (DeMarco. 1982) stated *"...you can't control what you can't measure"*. This clearly implies that measurement is as important for the SE discipline as it is for any other engineering discipline. Software metrics is used to measure attributes of software projects, processes or products. In addition, software metrics can also be used to identify and mitigate software project threats as well as reduce the total cost of development by taking remedial action early in the development process (Hall et al. 2005). Fenton and Pfleeger (Fenton and Pfleeger. 2002) defined measurement as: *" ...the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to a clearly defined rule..."*

In this Thesis, we measure the internal attributes of the systems to investigate inheritance. A distinction should be made between *internal* and *external attributes* of a product, process and project. Fenton and Pfleeger (Fenton and Pfleeger 2002) distinguish between internal and external attributes; *internal attributes* of a software system include size, coupling, and the amount of reuse used in a system and its *external attributes* include reliability, usability, and security of a system. Numerous studies were originally conducted to introduce metrics into SE. For example, the cyclomatic complexity metric of McCabe (McCabe 1976) measures programs based on their structural model. The Fan-in and Fan-out metrics of Henry and Kafura (Henry and Kafura 1981) measure the number of inputs and outputs of a given module, respectively. With the introduction of OO technology, more advanced suites of metrics were introduced (Chidamber and Kemerer 1994, Lorenz and Kidd 1994, Abreu and Carapuca 1994, Briand et al. 1999b, Rosenberg 1999, Arisholm et al. 2004, Harrison et al. 1998b, Briand et al. 1998).

Abreu and Carapuca (Abreu and Carapuca 1994) defined the MOOD (Metrics for Object-Oriented Design) set of metrics. The MOOD metrics provide an indication of quality of an OO system. Each metric in the MOOD set of metrics measures one distinct aspect of an OO system. The MOOD set of metrics comprised: Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (PF), Coupling Factor (CF), Clustering Factor (CF), and Reuse Factor (RF).

Chidamber and Kemerer (Chidamber and Kemerer 1994) also proposed a suite of metrics to measure features of OO systems. Their suite of metrics consisted of the Weighted Methods per Class (WMC), Response For a Class (RFC), Lack of Cohesion in the Methods of a class (LCOM), Depth of Inheritance Tree of a class (DIT), Number Of Children of a class (NOC) and Coupling Between Objects (CBO).

Lorenz and Kidd (Lorenz and Kidd 1994) introduced a set of eleven metrics for measuring OO systems and divided the metrics into following four main categories: size, inheritance, internals and externals. Their set of metrics included the following four inheritance related metrics: number of operations overridden by a class; measuring the number of overridden operations/method of a subclass, number of operations added by a subclass; measuring the total number of new operations/methods added in a subclass, specialization index; measuring the level of specialization for each subclass, and class size; measuring the size of a class by counting the total number of operations/methods, (both inherited and non-inherited) and total number of inherited and non-inherited attributes in a class. Since the main focus of our research was on inheritance, we required a set of well-defined and validated metrics to measure inheritance in the systems that formed our study.

Coupling and cohesion are also two important facets of the OO paradigm. Briand et al. (Briand et al. 1998) presented a framework for measuring cohesion in OO systems. The framework consisted of numerous metrics designed to measure cohesion in OO systems. In a later study, Briand et al. (Briand et al. 1999b) presented an additional framework for measuring coupling in OO systems. A general and accepted tenet is that coupling is a detrimental factor for software comprehensibility - excessive coupling may consequently introduce faults into a system (Briand et al. 1999b). Inheritance is also a form of coupling and claims exist to suggest that coupling through inheritance is more favourable, from a system comprehension perspective, than non-inheritance coupling. English et al. (English et al. 2007) proposed a set of coupling metrics to measure *friendship* mechanism (in C++), inheritance and other forms of coupling and claimed that there was a need for metrics to measure various forms of coupling in OO systems. Harrison et al. (Harrison et al. 1998a) empirically assessed two coupling related metrics - the CBO of Chidamber and Kemerer and the number of associations between classes NAS (here, number of associations is the number of connection lines between classes in a UML diagram (Rumbaugh et al. 1998)). The metrics were applied to five C++ systems and the data for the two metrics compared to determine their efficiency and effectiveness. In that study, coupling was found to be independent of software understandability. The authors of the study also discovered a strong relationship between CBO and NAS and stated that only one of these coupling metrics was needed to measure coupling in OO systems.

In Chapter 5 we investigated the movement and re-location of classes from both size and coupling perspectives and found that larger classes, given by number of methods metric of Lorenz and Kidd (Lorenz and Kidd 1994), and highly coupled classes, given by message passing coupling metric of Li and Henry (Li and Henry 1993), were more frequently relocated within their corresponding inheritance hierarchy than smaller classes and loosely coupled classes, respectively. Measuring coupling dynamically provides a finer-grained insight into coupling between classes. Arisholm et al. (Arisholm et al. 2004) describe how coupling can be measured dynamically and introduced a suite of dynamic coupling measures. In that study, static, size and dynamic coupling measures were compared - the authors claimed that dynamic coupling measures could be used to analyze the change-proneness of OO systems.

It is also important to take into consideration the theoretical and empirical validations of metrics when using them in practice. Theoretical and empirical validations play a fundamental part in the ‘success’ of any software metric (Shepperd 1995). Metrics validation is the process of examining whether a software metric is a true numerical representation of the measured attribute (Fenton and Pfleeger 2002). In the past decade, researchers seem to have shifted the focus of their research to investigating and validating existing software metrics rather than introducing new metrics. The framework introduced by Kitchenham et al. (Kitchenham et al. 1995) showed how software metrics should be investigated for validity. They suggested that the following characteristics of a metric should be considered when investigating its validity: the real world object (entity), the property of the entity (attribute), how the attribute can be measured (units) and what scale type to be used to measure the attribute (scale type). Harrison et al. (Harrison et al. 1998b) reported the results of an investigation into the MOOD set of metrics, taking into consideration the OO features (i.e., encapsulation, inheritance, coupling and polymorphism). They argued that the MOOD set of metrics could provide an overall assessment of a software system.

Software metrics have also been investigated from a fault-prediction perspective. For example, Ping et al. (Ping et al. 2002) reported a case study validating a set of ten OO metrics from a fault prediction perspective. The metrics used were related to size, coupling, cohesion, inheritance, and reuse in OO systems. Ping et al. also claimed that the two types of coupling (inheritance and non-inheritance coupling) had different implications on fault-proneness and hence should be treated differently. (In Chapter 6 we investigate the impact of method calls on cohesion of a class. We distinguished between method calls within inheritance hierarchy (JHawk 2008) and external method calls (JHawk 2008) and found that method calls of both sorts reduced cohesion in a class.) In El Emam et al. (El Emam et al. 2001) a fault-prediction model was presented. The main finding of the study was that inheritance and export coupling metrics were strongly associated to fault-proneness. Briand et al. (Briand et al. 2000) also empirically investigated the relationship between OO coupling, cohesion and inheritance measures and the likelihood of fault detection in a system. They found that most coupling and inheritance measures introduced in the literature were closely related to the likelihood of faults in a system. However, cohesion did not have a major impact on fault-proneness. In that study, it was also reported that the majority of measures introduced in the literature were redundant, capturing the same dimensions in the data set. The redundancy of metrics was also reported in (Briand and Wust 2002).

Software metrics can also be useful tools for managers' decision making and resource allocation. The research described in (Chidamber et al. 1998) investigated the suite of metrics proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994) from a managerial prospective. The authors of the study claimed that the suite of metrics investigated was useful for making decisions pertaining to resource allocation and maintenance cost estimation. Basili et al. (Basili et al. 1996a) argued that the Chidamber and Kemerer metrics were, firstly, better predictors of faults and secondly, could be collected early in the development process. Since our main focus in this Thesis will be on changes of inheritance and particularly changes at various inheritance levels, we selected the DIT metric of Chidamber and Kemerer as a basis of our analysis of inheritance.

The impact of size on fault-proneness was also investigated in a study of a large NASA C++ ground system by Counsell (Counsell 2008). It was reported that the size of a class, given by the number of methods, was the main contributing factor to faults. Furthermore, coupling, given by CBO of Chidamber and Kemerer, was also found to increase faults in a class and class coupling at lower levels of inheritance hierarchy was found to be the most "harmful" type of coupling.

2.2.3 Inheritance

Empirical evidence suggests that reuse improves productivity (Lewis et al. 1991). In Java, a class can inherit from only one other class by using the keyword 'extends'. Java uses interfaces to facilitate multiple inheritance. Interfaces are similar to abstract classes (from which instance objects cannot be created); however, the attributes in an interface are always declared as *static final* (i.e., indicating that there is only one copy of the attribute for all objects of the class in which the attribute is defined) and the methods are always abstract (i.e., have no implementation - the subclass of the class/interface in which the abstract method is defined should provide the implementation for the abstract method).

Inheritance and its use in practice has been a controversial research topic. It has been subject to numerous studies in the literature (Bieman and Zhao 1995, Daly et al. 1996, Harrison et al. 2000, Cartwright 1998, Cartwright and Shepperd 2000, Prechelt et al. 2003, Tempero et al. 2008). Bieman and Zhao (Bieman and Zhao 1995) claimed that the amount of inheritance used in the systems was far less than recommended. Only 37% of the systems studied contained a median value for class inheritance depth greater than 1 (in Java only class 'Object' has the inheritance depth value of 0, which is inherited by all application classes) - the inheritance trees in the systems studied were found to be '*shallow*'. Daly et al. (Daly et al. 1996) describe an experiment in which subjects were timed performing maintenance tasks on OO systems of varying levels of inheritance. Systems with 3 levels of inheritance were shown to be easier to modify than systems with no inheritance. Systems with 5 levels of inheritance were, however, shown to take longer to modify than the systems without inheritance. Their results implied that the use of inheritance at shallow levels made system maintenance easier. However, inheritance beyond level three could be a confounding factor for software maintenance. The experiment was replicated by Cartwright (Cartwright 1998) and Harrison et al. (Harrison et al. 2000). Cartwright found that subjects took considerably longer to make changes to the system containing inheritance. However, the changes made to the system tended to be short and concise. Harrison et al. also found that flat systems (i.e., containing no inheritance) were easier to modify than systems containing three or five levels of inheritance, although their results also indicated that larger systems were equally difficult to understand whether or not they contained inheritance. In a multi-method study, Wood et al. (Wood et al. 1999) investigated facets of OO paradigm and suggested that inheritance should be used with care and only when required.

In a controlled experiment, Prechelt et al. (Prechelt et al. 2003) compared the performance of maintenance tasks on three programs with similar functionality and various levels of inheritance (zero, three and five levels of inheritance). They argued that the less inheritance used in a program, the less time it takes to maintain that program. The implications for coupling, cohesion and inheritance on fault-proneness were investigated by Briand et al. (Briand et al. 2001). They collected OO metrics from a commercial system developed by software professionals. In that study, cohesion and inheritance were not found to be good quality indicators in the system. From a coupling perspective, their findings suggested that each type of coupling should be assessed individually (some types of coupling e.g., method invocation and import coupling were found to be better quality indicators than others). The view that classes located at deeper levels of inheritance, containing fewer faults than classes located higher up in the same hierarchy was also reported in (Briand et al. 1999c). In addition, contradictory results suggested that classes at deeper levels of inheritance were more fault-prone than classes at higher levels, casting doubt on the effective use of inheritance at deeper levels. Briand et al. (Briand et al. 2002) presented a study to build a prediction model using a set of OO metrics. The authors of the study claimed that classes at deep levels of the inheritance hierarchy tended to be more fault-prone than classes residing higher up. This view was also reported in another study by Briand et al. (Briand et al. 2000), where the relationship between design measures and system quality was investigated.

In a study of the validation of a set of twenty four OO metrics using a telecommunication system, Glasberg et al. (Glasberg et al. 2000) found that classes in the middle part of inheritance hierarchy contained more faults than classes located near the root or leaf. The results of our investigation (Nasseri and Counsell 2008) indicated that the majority of warnings, which may *potentially* generate faults in a system, were found where the majority of functionality resided, irrespective of class position in the class hierarchy. Cartwright and Shepperd (Cartwright and Shepperd 2000) described an empirical investigation of a large telecommunication C++ system. They found that there was a positive correlation between the DIT metric of Chidamber and Kemerer and the number of user reported problems, casting doubt on deeper levels of inheritance. They suggested that software developers should pay extra attention when using deeper levels of inheritance. In that study it was also found that the use of inheritance in the system studied was negligible. Tempero et al. (Tempero et al. 2008) presented an empirical investigation of use of inheritance and introduced a set of structured metrics for measuring different forms of inheritance (i.e., interfaces and/or classes) in Java systems. The set of metrics introduced and other traditional metrics were then applied in practice using a collection of ninety OSS. They emphasized that a distinction should be made between different forms of inheritance relationship (i.e., classes and interfaces through extends and implements relationships) when measuring inheritance in a system. Each type is used in different ways and for a different purpose. Contrary to what the other studies found (Bieman and Zhao 1995, Cartwright and Shepperd 2000), Tempero et al. also discovered that the use of inheritance in the systems studied was higher than expected. They therefore claimed that using inheritance was a common development practice in OO systems. Furthermore, it was also found that user defined classes were used in a different way than that of library and other third party classes. User defined classes were primarily found to be used for defining other user defined classes.

There is also evidence to suggest that inheritance is a closely related topic to encapsulation and hence they should be investigated concurrently (Snyder 1986, Skogland 2003). The preceding survey of the literature on inheritance has shown that a considerable amount of research has been dedicated to inheritance and its implications on system fault-proneness and maintainability. Unfortunately, in our preliminary literature review we found very little evidence, if any, of studies investigating inheritance from an evolutionary perspective. In other words, we have limited knowledge of how inheritance structures evolve. The lack of such studies therefore inspired us to conduct a thorough investigation of evolution of inheritance at different levels of granularity, and build up a body of knowledge into evolution of OO systems from an inheritance perspective.

2.2.4 Software Maintenance and Evolution

Software maintenance is the process of making changes to an existing system and software evolution refers to the process of development of a system and its continuous changes for improvement. Swanson (Swanson 1976) introduced the following three main categories of software maintenance. 1) *Corrective maintenance*: the set of changes made to a software module to correct faults. 2) *Adaptive Maintenance*: the set of changes made to a system imposed by the environment (i.e., business requirement, legal issues, etc) in which the system is operating and 3) *perfective maintenance*: the set of changes made to a system in order to add new functionality to the system. Dvorak (Dvorak 1994) defined software maintenance and evolution as: *"…the correction of errors, and the implementation of modification needed to allow an existing system to perform new tasks, and to perform old ones under new conditions… …software evolution is the dynamic behaviour of programming systems as they are maintained and enhanced over their life time."*

Software maintenance is the most costly phase of software development life cycle. Lehman (Lehman 1980b) reported that in the US in 1977, approximately 70% of total cost of software was on software maintenance. According to Meyers (Meyers 1988) software maintenance accounts for 60% to 85% of overall cost of software. Software researchers have therefore dedicated numerous studies to identify and mitigate the problems associated with the process of software maintenance and evolution (Daly et al. 1996, Arisholm and Briand 2006, Lehman et al, 1997, Arisholm et al. 2007, Deligiannis et al. 2003, Sangwan et al. 2008).

Buckley et al (Buckley et al. 2004) claimed that an appreciation of the information needs of programmers maintaining a system can help reduce the overall maintenance cost of the system. They proposed an approach, called *content analysis*, to help ascertain information needs of programmers and effectively communicate that information with those programmers when maintaining a system. Basili (Basili 1990) described the following three models of software maintenance.

> "*1) Quick-fix model: Quick-fix model presents an abstraction of typical approach to software maintenance. In this model, you take an existing system, usually source code and make necessary changes to the code and relevant documentation, and compile the system as a new version.*
> *2) Iterative-enhanced model: this is an evolutionary model proposed for development where the full requirement of the system has not been understood. It starts with the existing system's requirement, design, code, test and analysis documentation. It uses the reusable components of the existing system to build the new system.*
> *3) Full-reuse model: Full-reuse model starts with the requirement analysis and design of the new system and reuses the appropriate requirements, design, and code from an earlier version and similar systems in the system repository.*"

In a later study, Basili et al. (Basili et al. 1996b) presented the results of an experiment conducted at University of Maryland assessing the impact of reuse into defect density, software productivity, rework and effort in OO systems. They claimed that reuse in OO systems reduced defect density and resulted in lower rework. Furthermore, they also argued that reuse also increased productivity and reduced development effort. In a study of software evolution, Lehman (Lehman 1974) introduced the laws of software evolution which were then revisited for amendments in subsequent studies (Lehman 1978, Lehman 1980a, Lehman 1996). In a later study, Lehman (Lehman 1980b) described computers and programs, and how they were used in practice. The programs were placed into following three categories: 1) S-Programs: programs the specification of which can provide its function. 2) P-Programs: programs that are the representation of real world situation and are unpredictable. 3) E-Programs: constantly changing programs. The author then investigated the laws of program evolution using eighteen versions of an operating system. It was suggested that software planning should not entirely depend on business requirements. Other factors such as dynamic characteristics of the process and system should also be taken into account. Moreover, Lehman et al. (Lehman et al. 1997) empirically investigated the laws of software evolution, and compared the results to studies conducted in 1970's (Belady and Lehman 1972, Lehman 1974). Their results supported the laws of software evolution and suggested that, despite the 20 years time gap, the 1970's approach to metrics analysis was still pertinent to software evolution. Lehman's laws of software evolution provide an insight into overall system evolution.

Software process planning and management is an important part of the successful evolution of any software. In Lehman and Ramil (Lehman and Ramil 2001) fifty recommendations for supporting software process planning and management were introduced. They claimed that the recommendations introduced could easily be embedded into tools to support software process planning and management. Bergel et al. (Bergel et al. 2005) described a study presenting the notion of Classbox model (*…a module that restricts the visibility of changes to selected clients only…*) in statically-typed languages including Java. They presented a case study illustrating how classboxes/j (a prototype implementation of classbox for Java) was employed to provide a better implementation of Swing (a GUI system). They claimed that classboxes provided a better approach to unanticipated modification over a system and could also be used to determine the impact of those changes. Kemerer and Slaughter (Kemerer and Slaughter 1999) presented a longitudinal study introducing new methods and techniques for studying software evolution. 25000 change events from 23 commercial systems over 20 years time span were collected and analyzed. The authors used and adapted the existing methods and techniques to study evolution of two representative systems. They designed a new approach for conducting longitudinal studies so that software evolution could be analysed at different levels of granularity (system and module). In (Nasseri and Counsell 2009a, Nasseri and Counsell 2009b) we observed that analyzing a system at a finer-grain (i.e., method and attribute level) can often identify the changes that may go undetected when analyzing the same system at higher granularity (i.e., class and package level).

Girba et al. (Girba et al. 2005) presented a study characterizing the structural evolution of a system. They proposed the history of code as *first-class* entity for characterizing class hierarchies and suggested measurements to precisely measure class hierarchies through multiple versions of a system. The authors deduced that historical data played a key part when studying the evolution of class hierarchies. Buckley et al. (Buckley et al. 2005) proposed a taxonomy of software evolution which characterised change dimensions into the following four themes: *when, where, what* and *how* changes are made to a system. In

Girba and Ducasse (Girba and Ducasse 2006) a need for an explicit meta-model for software evolution analysis was stressed. They presented a sequence of requirements essential for evolutionary meta-model and presented a meta-model called 'Hismo'. The Hismo meta-model emphasized on the history of a system, the time, and structural entities when analyzing software evolution. With the growing popularity of OSS in the SE research community, OSS development itself has undergone protracted research. One possible explanation for this can be that OSS are easily and freely accessed (Capiluppi et al. 2004, Capiluppi and Ramil 2004, Counsell and Swift 2008). In a study of four OSS, Counsell et al. (Counsell et al. 2006b) investigated the role of inner classes in errors made by manual data collections and assessing the role of class size on those errors. They measured the size of a class in terms of number of attributes and methods defined in that class and claimed that class size and number of inner classes had no significant impact on the number of errors made in manual data collection.

Counsell and Swift (Counsell and Swift 2008) empirically investigated the trends that may exist in *potential-faults* (p-Faults) in OO software systems. p-Fault data was collected from ten Java OSS, using the FindBugs tool (FindBugs 2008). FindBugs extracts six categories of p-Faults including the code vulnerability p-Fault. They showed that firstly, the majority of classes with vulnerability p-Faults contained no other forms of p-Fault. Secondly, an association was identified between the code vulnerability p-Fault and '*Bad Practice*' (a category of p-Faults extracted by the FindBugs) p-Faults. We used the FindBugs tool to extract the six categories of p-Faults/warnings (in this Thesis we use the term 'warnings' as opposed to p-Faults used in (Counsell and Swift 2008)) and extrapolated the trends in warnings extracted from multiple versions of four OSS (Nasseri and Counsell 2008).

In an investigation of evolution of an OSS, Capiluppi et al. (Capiluppi et al. 2004) found that the system studied grew in terms of number of files, folders, lines of code, source lines of code and kilobytes. Secondly, the tree-like structure of folders tended to grow '*breadth-wise*' rather than '*depth-wise*' (This finding was also confirmed in our study (Nasseri et al 2008) that inheritance hierarchies tended to grow breadth-wise rather than depth-wise). In a further study of two OSS, Capiluppi and Ramil (Capliluppi and Ramil 2004) claimed that the evolutionary attributes of the two systems manifested some similarities at a high level of abstraction. In addition, the two systems, due to the different characteristics also showed discrepancies in their evolutionary behaviour. The evolution of one of the systems was found to be adaptable (exhibiting a high growth rate), while the evolution of the other was found to stagnate in certain releases and did not exhibit a significant growth rate.

Stamelos et al. (Stamelos et al. 2002) reported the results of a pilot study assessing the structural quality of code in OSS. They argued that the quality of the code delivered by OSS was lower than expected by the industrial standard which however, had potential for further improvements. Stamelos et al. also argued that OSS may require the definition of their own quality standard suggesting that the nature of OSS development should be taken into account when defining its quality.

The general tenet also holds that software complexity increases as a system evolves unless work is done to impede its rise. In a relatively recent study, Sangwan et al. (Sangwan et al. 2008) investigated the structural complexity of three OSS as they evolved. They argued that as the studied systems evolved, their structural complexity moved either from lower design structural levels to higher levels or from higher levels downwards. In the Thesis presented, we investigated the changes of inheritance in the form of new classes, deleted classes and moved classes in four OSS through multiple versions and found that classes were often moved or relocated from one level to another within the hierarchy (Chapter 5 and Nasseri et al 2009).

2.2.5 Software Refactoring

Since we are interested in changes of inheritance we believe one way to explore the structural changes of a system is to extract the *refactorings* applied to that system. Software refactoring is yet another technique which may challenge the inheritance structure of a system. Extracting refactoring data from a system can help observe the continuous changes made to that system. The term refactoring was initially used by Opdyke and Johnson (Opdyke and Johnson 1990), referring to the structural improvement of a system. In Fowler’s text (Fowler 1999) software refactoring was defined 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”*. Refactoring/restructuring is used to improve program comprehensibility and ease system modification, which therefore promotes software maintenance and reuse (Johnson and Foote 1988, Chikofsky et al. 1990). Software refactoring, due to its major role in software maintenance, has received a great deal of attention from SE research community (Fowler 1999, Opdyke and Johnson 1993, Kerievsky 2004). In the PhD work of Opdyke (Opdyke 1992), various types of refactorings were introduced. Opdyke's work focused on introduction of refactoring related to inheritance and aggregation. In a later study, Opdyke and Johnson (Opdyke and Johnson 1993) showed how to form an abstract class from an existing concrete class using refactoring. In that study, the process of refactoring was broken down into a sequence of refactoring steps and an approach was presented for automation of those refactoring steps. Johnson and Opdyke (Johnson and Opdyke 1993) introduced several refactorings including conversion of inheritance into aggregation and how to re-organise aggregation and inheritance hierarchies in a system by shifting variables and methods between classes.

Tokuda and Batory (Tokuda and Batory 2001) presented a study showing how refactoring can be carried out in an automated form. They carried out an automatic refactoring of fourteen thousand lines of code. Mens and Tourwe (Mens and Tourwe 2004) carried out an investigation of refactoring and the motivation for refactoring code *smells* (a code smell is an indication of a problem in a segment of code (Fowler 1999)). They reported that the application domains played a significant part when identifying the type of refactorings to be carried out. Counsell et al (Counsell et al. 2003) empirically investigated the trends of changes in fifty-two Java library classes over a period of three years. They showed that a significant number of changes were made at the method signature level, nearly the same amount of addition was found at the method call level. They also claimed that conscious refactoring is generally not undertaken by developers.

In (Counsell et al. 2006a), an analysis of seven OSS from a refactoring perspective was reported. They showed that there were six commonly used or 'core' refactorings among fifteen extracted from the seven systems. They also argued that in the seven systems studied, very few inheritance and encapsulation based refactorings were used. In our research we also use the refactoring data extracted in (Counsell et al. 2006a) and compare them to the method an attribute changes in those systems (Chapter 4 and Nasseri and Counsell 2009a, Nasseri and Counsell 2009b). Demeyer et al. (Demeyer et al. 2000) conducted a study introducing a set of four heuristics to detect addition/deletions or refactorings in different versions of a system. They argued that their approach was highly effective when conducting reverse engineering in OO systems.

The relationship between refactoring and unit testing was investigated by van Deursen and Moonen (van Deursen and Moonen 2002). They introduced a testing taxonomy and stressed the use of *test-first* refactoring (this type of refactoring takes into consideration existing test cases) as a starting point when undertaking refactoring. In an empirical study of seven Java systems, Advani et al. (Advani et al. 2005) investigated whether software systems undergo refactoring activity? If so, what types of refactorings were commonly undertaken? They claimed that firstly, simple refactorings were most commonly undertaken by developers. Secondly, refactorings predominantly occurred in the middle versions of a system not in earliest and/or latest versions. Finally, the number of refactorings related to inheritance was found to be relatively low. In a later study, Advani et al. (Advani et al. 2006) described an automated tool for collection of refactoring data from multiple versions of a system. The tool was designed to extract refactoring information from Java systems.

In our research, we empirically compared the changes of methods and attributes within inheritance hierarchies and refactorings carried out through multiple versions of four Java OSS (see Chapter 4). The refactoring data used was extracted using an automated tool, details of which can be found in (Advani et al. 2006).

2.3 Methodology Adopted

In this section, we present the methodology adopted for our research including a discussion of the available research methods in SE (Section 2.3.1); Section 2.3.2 presents our research design including forming research objectives, formulating hypotheses, sample selection, sampling procedure and criteria for choosing the subject systems, a description of the subject systems, data collection, definition of the software metrics used and statistical techniques employed.

2.3.1 Research Methods in Software Engineering

There are two research approaches available that can be adopted when conducting research. 1) qualitative research and 2) quantitative research. Qualitative research is concerned with people's views and attitudes in the form of non-numerical data towards a particular research question (Seaman 1999). It is often conducted using interviews, observations and questionnaires. Quantitative research is concerned with the examination of numerical forms of data in order to answer a particular research question (Perry et al. 2000). The selection of research approach depends on the context of research question and preference of researcher called the *research methodology*. In SE, research is conducted to improve the current situation of software, inform the SE community of any problems that may be associated with contemporary software development and introduce solutions to those problems. Fortunately, research in SE is versatile. It can be conducted using either qualitative or quantitative or in amalgamation (Wood et al. 1999).

For the research presented in this Thesis, we adopted a quantitative approach. We conducted a product based analysis using a set of eight Java OSS. Our research is centred around inheritance features of OO systems from an evolutionary perspective.

2.3.2 Research Design

Empirical studies in SE often use techniques to conduct research in the form of case studies, experiments, and surveys to empirically scrutinize a new phenomenon or replicate an already investigated phenomenon. However, the validity of the results is a key concern in empirical studies. The results of a study, using any method (whether case study, experiment and/or survey methods), should be valid from several perspectives. Researchers must take into consideration the *construct* validity (i.e., the degree to which the measured concept can be measured accurately in a different way), *internal* validity (i.e., demonstrating the dependency between independent and dependent variables), and finally, *external* validity (i.e., the degree to which the results of the study can be generalized). In addition, replication plays a key part in the validity of research outcomes. It determines the level of confidence that a reader should have in the outcomes of an investigation on a particular research question (Fenton and Pfleeger 2002).

In this research, we used product archived analysis using multiple versions of eight Java OSS. Our technique can be claimed to be similar to multiple case studies. However, it is different from experiments and survey methods. We used a static analysis of source code of eight Java OSS to extract OO metrics from versions of the systems using an automated tool described in Section 2.3.2.6.

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 inheritance which in turn provides an insight into the evolution of inheritance. Using this approach, we were able to reveal patterns of change in multiple versions of the studied systems, which was not possible by using a single case study, experiment or survey method. In addition, we observed that in the past researchers have used relatively small systems to experiment the maintainability of inheritance (Daly et al. 1996, Cartwright 1999, Harrison et al. 2000). We believe that analysing real-world large systems can bring to light the characteristics of inheritance that may go undetected when analysing a small system.

2.3.2.1 Forming research objectives

Identifying research objectives is a significant part of any research project. It helps clarify research direction on a step-by-step basis. After the research problem has been identified the next phase of research project management is to identify a set of goals to be accomplished.

2.3.2.2 Formulating hypotheses

In this Thesis we tested a set of hypotheses relating to inheritance, cohesion, coupling and class size, presented in Chapters 5 and 6. Speculation is an important aspect of an investigation. Hypotheses can help researchers speculate on the expected results and the direction of their investigation. During the process of generating hypotheses, a researcher must provide a set of ground basis indicating why they believe that the hypothesis will be supported, given by the available data. In other words, the theoretical aspects of a hypothesis should be taken into consideration when formulating a hypothesis.

Furthermore, hypothesis testing also requires identification of appropriate data closely related to *cause* and *effect* of the hypothesis. The data should be categorized 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 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 determined, an appropriate statistical test should be identified to scientifically test the impact of independent variable on the dependent variable(s). Hypotheses consist of a *null hypothesis* and an *alternative hypothesis*. A null hypothesis speculates that an independent variable has no significant relationship with dependent variable(s), and an alternative hypothesis speculates that a correlation exists between independent and dependent variables (Field 2006). Researchers speculate that the alternative hypothesis is true, unless the null hypothesis indicates the opposite.

2.3.2.3 Sample selection

Generalization of results is an increasingly important issue in empirical studies. Sampling is the process of selection of subjects/objects for a study. Researchers should adhere to scientific sampling when conducting empirical studies. It improves the generalizability of the outcomes of their investigation to a wider context (Kitchenham et al. 2002). There are two main designs of sample selection, 1) probabilistic and 2) non-probabilistic. Probability sampling uses an *arbitrary* or *random* selection of samples while non-probability sampling uses a *non-random* selection. This means that non-probability design sets a selection criterion by identifying the population which the samples represent. The population of a sample is a group(s) of people to which the results may be generalized. However, the main challenges that researchers face, when defining samples, is identification of the population elements, and selection of a representative sample to represent that population. When conducting artefact based empirical studies, it is impossible to access and analyze all available software artefacts and include them all in a single study. For generalization purposes, researchers often use the non-probability design by which they define a set of selection criteria, identify the population of interest and select a sample representing the population.

2.3.2.4 Sampling procedure and criteria for choosing the subject systems

For this Thesis, we opted to use a non-probability sampling design to carefully select a set of eight Java OSS. We justify the selection of non-probability approach by the fact that it was unfeasible for us or any other researcher to include all available systems in only one study. Researchers, due to the restrictions of time, cost and accessibility of software systems, should limit their selection and explicitly describe their inclusion and exclusion criteria for the systems they are using. The non-probability *purposive* approach was therefore used to define the following explicit criteria for the selection of our subject systems. They all have to be entirely Java systems. The systems should be real, not experimental. Sufficient versions are available (for a longitudinal study). The systems must consist of different sizes (in number of classes), and belong to a mix of various application domains (i.e., GUI applications, game engine, application Server etc). Systems were selected in 'number of downloads' order from sourceforge.net. The process of selection thus resulted in many systems being rejected from candidate systems identified from those listed in sourceforge.net because they were either a mix of different languages and/or did not contain multiple versions for download.

Our selection of system samples can also be justified on the basis that five of the eight systems used in our research were also subject to a previous empirical study (Advani et al. 2006) and three of those five systems were also used in two previous studies (Advani et al. 2005, Counsell et al. 2006a). The three studies analysed those systems from a refactoring perspective and our research is concerned with the changes in inheritance. We therefore believe that our findings can inform those of previous studies. We justify our decision to select the systems from multiple application domains by the fact that it would enable us to generalize our findings into a broader OO population.

2.3.2.5 Description of the subject systems

The following is a description of the subject systems. The eight systems in ascending order of number of versions are as follows:

1. HSQLDB: a relational database engine implemented in Java. This system comprised 6 versions. HSQLDB started with 56 classes in first version and comprised 358 classes by the final version.

2. JColibri: an OO framework in Java for Case-Base Reasoning (CBR) system. JColibri comprised 8 versions. It started with 179 classes in version 1 which increased to 417 classes by version 7 and dropped to 228 Java classes in version 8.

3. JasperReports: a business intelligence and reporting engine. This system comprised 12 versions. JasperReports started with 818 classes in version 1 and comprised 1098 classes by the final version.

4. EasyWay: a 2D Java game engine. This system comprised 21 versions. EasyWay started with 183 classes in version 1 and comprised 197 classes by final version.

5. SwingWT: an implementation of the Java Swing and AWT APIs. This system comprised 22 versions. SwingWT started with 50 classes in its first version and increased in size to 620 by the final version.

6. JAG: Java Application Generator. Generates working projects containing complete J2EE applications. This system comprised 23 versions. JAG started with 137 classes in its first version and contained 136 classes by the final version.

7. JBoss: a standards-compliant, J2EE based application server implemented in Java. 27 versions of this system were available starting from version 8. JBoss was the largest system in size. It contained 3934 classes in version 8 and variably evolved to 9082 classes by the version 34 (its final version when accessed). The size of JBoss exceeded 10000 classes in versions 27, 30 and 33.

8. Tyrant: a graphical fantasy adventure game. 45 versions of this system were studied. Tyrant started with 122 classes and finally ended with 273 classes by the final version.

2.3.2.6 Data collection and metrics definition

It is important for a researcher to analyze whether the software metric used is well defined and valid (Fenton and Pfleeger 2002). This analysis ensures whether the software metric(s) actually measures the attribute(s) of a product, process or project which it claims 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 described in Section 2.3.2.5. It uses static analysis of source code to extract numerous OO metrics in the literature. We justify our selection of the tool on the basis that it was used and recommended to us by other researchers in the field of SE (Arisholm and Briand 2006, Arisholm et al. 2007). The following is a description of the metric definitions used throughout this Thesis:

1. Depth of Inheritance Tree (DIT): this metric measures the number of ancestors of a class including ‘Object’ from which all classes inherit. The DIT metric was proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994). We assume the value of DIT for class ‘Object’ at the root of the entire hierarchy is zero; hence, all classes declared at level 1 implicitly *extend* only class ‘Object’.
2. Specialization Ratio (SR): this metric is calculated as: the number of subclasses of a class divided by the number of its superclasses. High values of the SR metric imply high level of reuse through

subclassing. The SR metric was proposed by Henderson-Sellers (Henderson-Sellers 1995).

3. Reuse Ratio (RR): this metric measures inheritance using the formula: number of superclasses of a class divided by the total number of classes. The total number of classes refers to total number of classes residing in inheritance hierarchy excluding class ‘Object’. The RR metric was proposed by Henderson-Sellers (Henderson-Sellers 1995). An RR Value close to 1 implies that the inheritance hierarchy is *narrow*. An RR value close to 0 implies that the inheritance hierarchy is *shallow*.

4. Number of Children (NOC): this metric measures the number of immediate subclasses of a class and was proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994).

5. Number of Methods (NOM): this metric measures the total number of methods in a class. The NOM metric is that proposed by Lorenz and Kidd (Lorenz and Kidd 1994) and is similar to that of WMC (Weighted Methods per Class) of Chidamber and Kemerer (Chidamber and Kemerer 1994).

6. Number of Attributes (NOA): this metric measures the total number of local variables plus the total number of class variables (public, private and protected). The number of attributes metric is that proposed by Lorenz and Kidd (Lorenz and Kidd 1994).

7. Number of calls to methods within Hierarchy (HIER): this metric measures the number of method calls that are in class hierarchy for a class JHawk (JHawk 2008). For example, in Figure 2.1, MethodY in ClassB calls MethodX defined in its superclass (ClassA) is a call to a method in the hierarchy.

8. Number of External method calls (EXT): EXT metric measures the number of method calls in a class to methods of other classes JHawk (JHawk 2008), excluding HIER calls. For example, in Figure 2.1 MethodY in ClassB calls MethodZ in ClassC which is not in the same class hierarchy (the external method calls excludes class Object inherited by every class in Java).

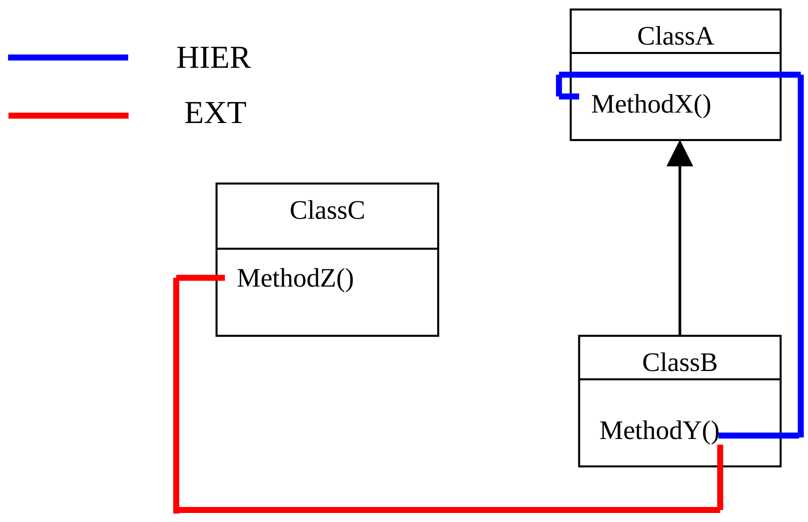


Figure 2.1. An example of HIER and EXT metrics

9. Lack of Cohesion Of the Methods in a class (LCOM): LCOM measures the relations of methods and local variables of a class by counting the number of method pairs accessing different fields/variables minus the number of method pairs accessing the same fields/variables. The LCOM metric is that proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994). A high LCOM for a class is undesirable and indicates high complexity in that class.

10. Message Passing Coupling (MPC): The MPC 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 on the methods of other classes. This includes both HIER and EXT. The MPC metric is that proposed by Li and Henry (Li and Henry 1993).

2.3.2.7 Metrics selection criteria

In this Thesis, we are interested in inheritance based metrics. We aim to measure inheritance at various levels of granularity (from class to method and attribute levels). The explicit criteria for measuring inheritance were that the metrics should be related to inheritance and should measure the position of a class within the hierarchy from a depth perspective. We therefore opted to use the DIT as a core metric to measure class location within the inheritance hierarchies. Since no metric has been introduced to measure the width of inheritance hierarchies, we inevitably used the DIT and the number of classes at each level of an inheritance hierarchy instead. Furthermore, we measured the amount of reuse in a system; we opted to collect the SR, RR and NOC metrics. To measure class size we collected the number of methods and attributes metrics. Cohesion was measured using the LCOM and coupling was measured using the MPC, HIER and EXT metrics

2.3.2.8 Statistical techniques

In this Thesis, we used three correlation coefficient analyses (Pearson's, Kendall's and Spearman's) to investigate the relationship between the size of a class, given by NOM, and its interaction with other classes, give by the HIER and EXT (Chapter 6). Only two correlation coefficient analyses (Kendall's and Spearman's) were employed to investigate the impact of size and coupling of a class, given respectively by NOM and MPC, on cohesiveness of that classes, given by LCOM (Chapter 5). In terms of other tests, we used Mann-Whitney U-tests to investigate the difference between the cohesion of moved (moved within the hierarchy) and static classes in four systems (Chapter 5). The Mann-Whitney U-test was also used to investigate the difference between the cohesion of the classes containing method calls and classes without any method calls (Chapter 6). Furthermore, we employed the Wilcoxon signed-rank test to investigate the impact of class movement and re-location on class cohesion (Chapter 5).

Correlation analysis is a statistical test which examines the linear inter-dependency of the changes in one variable on the changes of another variable (Cohen et al. 2003). The difference between the types of correlation techniques is that Spearman's and Kendall's correlation coefficient analyses are applicable when the data is non-parametric (where the data assumes a normal distribution); whereas Pearson's correlation is pertinent to a parametric set of data.

A Mann-Whitney U-test is an analysis of the differences between the distributions of two samples (Hinkle et al. 1995). The Mann-Whitney U-test is an alternative to the two-sample student’s t-test when the criteria set for the t-test is not met. The Mann-Whitney U-test requires the data to have an ordinal scale and be non-parametric. The Wilcoxon test is also a non-parametric test equivalent to student’s t-test and is used when the criteria set for the t-test is not met (Wilcoxon 1945). The Wilcoxon test is used for the case of two related samples or repeated measurements on a single sample.

2.4 Summary

In first part of this chapter, we provided an overview of the related work in the area of empirical SE, software metrics, inheritance, software maintenance and evolution and refactoring. We also briefly explained how those works are related to our research. In the second part of the chapter, we presented a discussion of the methodology used to conduct our empirical research including, the design of the study, a description of the sample systems selected, a justification of our sample selection, the definition of software metrics used, criteria for selection of software metrics, data collection and statistical techniques used.

The following chapters will present a thorough empirical investigation of trends of changes made to inheritance hierarchies in Java OSS. The investigation helped us model the changes of inheritance and build a body of knowledge of evolutionary behaviour of the systems from an inheritance perspective. We examined the changes of inheritance at various levels of granularity from a class to method and attribute level.

CHAPTER 3 Inheritance and Change in Java OSS

3.1 Introduction

In the previous chapter we provided a survey of related work in literature and the methodology used to conduct the empirical research in this Thesis. Previous studies of OO software have reported avoidance of the inheritance mechanism and have cast doubt on the wisdom of 'deep' inheritance levels. From an evolutionary perspective, the picture is unclear - we still know relatively little about how, over time, changes tend to be applied by developers. Our conjecture is that an inheritance hierarchy will tend to grow 'breadth-wise' rather than 'depth-wise'. This claim is made on the basis that developers will avoid extending depth in favour of breadth because of the inherent complexity of having to understand the functionality of superclasses. Thus the goal of our study is to investigate this empirically.

In this chapter, we present an empirical study of seven Java OSS over a series of versions to observe the nature and location of class changes (additions and deletions) within the inheritance hierarchies. In addition, we investigate the changes of inheritance at a finer-grain (at the method and attribute level) in a subset of the seven systems (the same changes of methods and attributes for the remaining four systems in our system archive is presented in Chapter 4, and a more detailed analysis of changes of inheritance, in terms of new classes, deleted classes, and moved classes, in four of the seven systems is given in Chapter 5). This study is of significance for two reasons. Firstly, if we can predict the most change-prone parts of a system then we can pre-emptively target refactoring activity to such parts of a system. Secondly, it may yield information as to how software engineers view and understand complex legacy systems. The research problem is: how do inheritance hierarchies in OO software systems evolve over time? More specifically, we conjecture that 'change' will not be evenly distributed but will tend to cluster around the top levels (closer to the root) of such structures**.**

In Section 3.2, we present the motivation for our empirical investigation, and related issues. Section 3.3 describes the details of the empirical study, including a description of the seven systems used, data collected and summary data. In Section 3.4, we present the data analysis at class, method and attribute level. We then present a discussion of the results (Section 3.5) and, finally, in Section 3.6, we present a summary of the empirical study and conclusions. Part of this chapter was originally published in (Nasseri et al. 2008).

3.2 Study Motivation

The original claim for using inheritance was that it modelled data in a structured and logical fashion, thus aiding the maintenance process (Booch 1993). The motivation for the study in this chapter stems from a number of sources. Firstly, we know very little about how inheritance structures evolve over time; the investigation in this chapter seeks to shed light upon this issue. There is evidence to suggest that developers may find inheritance difficult to comprehend beyond a specific level (Daly et al. 1996, Cartwright and Shepperd 2000, Harrison et al. 2000). If that is true, then we would expect developers to add classes at shallow levels of the inheritance hierarchy rather than at deep levels. We posit that growth will be breadth-wise not depth-wise, thus supporting a growing belief about the use of inheritance. We believe that a better understanding of the change behaviour, and in particular the locality, would enable refactoring resources to be targeted more efficiently. Secondly, we believe that a first-step towards a change prediction model is an appreciation of current trends in changes made to an inheritance hierarchy. Given that this is a resource intensive activity, this would clearly be of benefit to software engineers (and potentially users) since the outcome could be more flexible and responsive software systems.

We present an empirical investigation of trends of inheritance in the evolution of seven of the eight Java OSS (when this study was conducted only seven systems were available in our system archive) presented in Section 2.3.2.5. Inheritance-based data was collected and examined from the systems on a version-by-version basis, to extrapolate the trends of changes of inheritance structure in the systems studied. Furthermore, we investigate the changes of methods and attributes in a subset of the seven systems.

3.3 Study Details

3.3.1 The seven open-source systems

The seven OSS on which our study is based were HSQLDB, JasperReports, EasyWay, SwingWT, JAG, JBoss and Tyrant (see systems 1, 3, 4, 5, 6, 7 and 8 in Section 2.3.2.5). For this study, we included all available versions of the seven systems. We note that the 'final' version represents the latest version available to download and not the end version of the systems.

3.3.2 Data Collected

For this study, we used the JHawk tool (described in Section 2.3.2.6) to collect inheritance-based measures from each version of the seven systems. The inheritance metrics collected were as follows: Depth of Inheritance Tree (DIT), Specialization Ratio (SR), Reuse Ratio (RR), and Number Of Children (NOC) (see metrics 1, 2, 3 and 4 in Section 2.3.2.6). In addition, we collected the following size metrics: Number Of Methods (NOM) and Number Of Attributes (NOA) (see metrics 5 and 6 in Section 2.3.2.6) from a subset of the seven systems (EasyWay, JAG and JBoss) to discover the finer-grain changes of classes at each DIT level. The measures were collected from classes of each version of the seven systems. Note that we refer to a single 'inheritance hierarchy' of Java throughout the chapter, since in Java every class inherits from 'Object'. This is distinct from C++ where a class need not necessarily inherit from any other class. We also make no distinction between concrete class, abstract class and interface for the purposes of our analysis.

3.3.3 Summary Data

Table 3.1 shows, for each of the seven systems, in order of versions studied, the maximum (Max), minimum (Min), median (Med) and Mean change values in the number of classes across the versions studied. By 'change' we mean positive or negative 'growth' by either addition or deletion of classes. For maximum change, Table 3.1 also indicates the normalized (Norm.) percentage of Max. to indicate what percentage of initial system size that Max. change represents. For example, the Max. change of 176 for the HSQLDB represented an increase of 271% in that system over its original size (of 56 classes). We also include the approximate variance (Var.) values for the set of changes for versions of each system. For example, the variance of the set of changes from version to version of the HSQLDB system was 15336.

System	Max. Ch	Norm.	Min. Ch	Var.	Med. Ch	Mean Ch
HSQLDB	176	271%	0	15336	23.5	58.6
JasperReports	183	22%	-77	11696	13.5	23.3
EasyWay	16	9%	-18	190	0	0.76
SwingWT	160	320%	0	39327	20.5	27.19
JAG	3	2%	-12	17	1.0	1.0
JBoss	4537	115%	- 4506	5073056	245	476.9
Tyrant	103	84%	-85	1657	0	3.58

Table 3.1. Summary class change data for the seven systems (all versions)

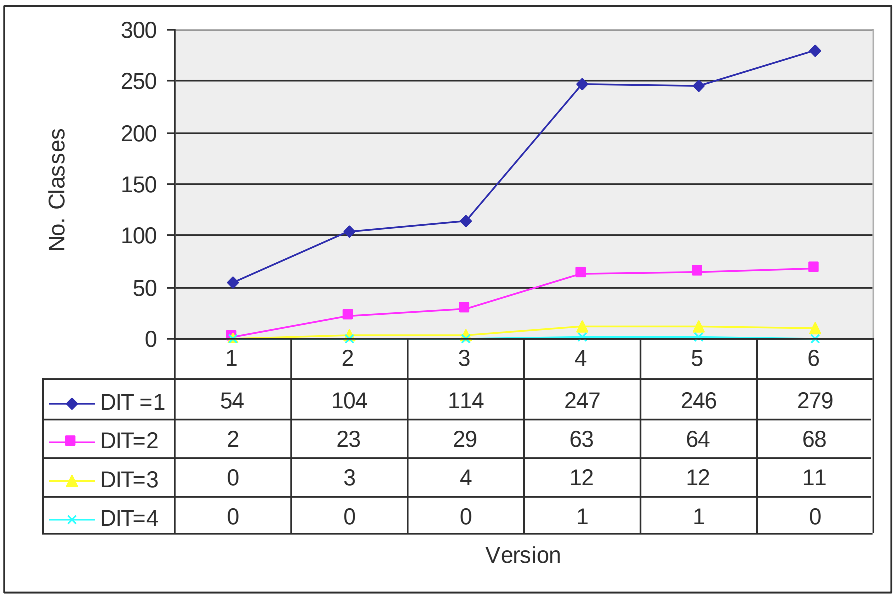
From Table 3.1 we see considerable variation in the behaviour of the systems. However, the mean change is always positive indicating a tendency to grow in size over time. This is most pronounced for JBoss. The size of a release or change is also most erratic for JBoss according to its variance. The EasyWay, JAG and Tyrant systems all have relatively low median and mean change values. If we consider a *stable* system as one with a ‘close to zero mean change value and low variance’ then although no single system satisfies these criteria, JAG and EasyWay seem the most stable of our seven systems. Remarkable is the fact that Tyrant contained twenty-three ‘transitions’ from one version to the next, where no change in the number of classes was noted (and hence could be considered the most stable of the seven systems even though it does not have the smallest variance of the systems studied). It is also worth noting that the number of versions studied is not a particularly good indicator of size of change. One of the lowest mean changes belongs to Tyrant and the second largest mean change belongs to HSQLDB. If we view stability through the Norm. values from Table 3.1, then the JAG and EasyWay systems figure prominently again (as does the JasperReports system).

3.4 Data Analysis

Our analysis now considers the evidence to support our conjecture that the inheritance hierarchy grows in ‘breadth’ rather than ‘depth’. We begin with a coarse-grained analysis of the trends in numbers (i.e. frequency) of classes at each DIT level on a version-by-version basis for each of the seven systems.

3.4.1 Coarse-grained DIT analysis

Figure 3.1 gives the frequency of DIT values for classes in the versions of HSQLDB and shows (apart from DIT level four) a strong tendency for classes to be consistently added (i.e., representing a net increase) at DIT levels one, two and three throughout. There is particularly strong evidence of classes being added at DIT levels one and two of the hierarchy. There is only one single class at DIT level four and this class disappears by version 6. The strength of addition at DIT level one is illustrated by the fact that of the 302 classes added to this system over the course of the 6 versions, 225 were added to DIT level one and 66 added to DIT level two. Combined, this represents 96.36% of the total. Only 11 classes were added to DIT level three. Thus we have a system that is characterized by change at shallow levels of the hierarchy.



	1	2	3	4	5	6
◆ DIT =1	54	104	114	247	246	279
■ DIT=2	2	23	29	63	64	68
▲ DIT=3	0	3	4	12	12	11
× DIT=4	0	0	0	1	1	0

Figure 3.1. DIT frequencies HSQLDB (all versions)

Figure 3.2 shows the same breakdown of the frequency of DIT values for versions of JasperReports and shows a similar upward trend to that of Figure 3.1. It appears that, again, the majority of classes were added at DIT levels one and two. Interestingly, the number of classes at levels four and five (10 and 4, respectively) did not change throughout the entire set of 12 versions studied. Of the 280 net classes added to JasperReports, only 13 were added to DIT level three. In contrast, 267 classes, representing 95.36% of the total were added to DIT levels one and two.

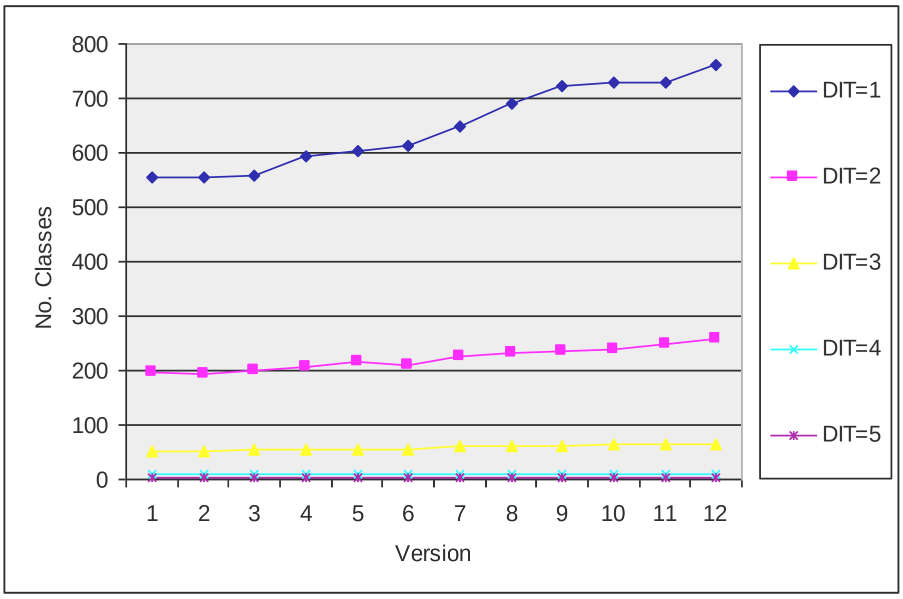


Figure 3.2. DIT frequencies JasperReports (all versions)

Figure 3.3 shows the frequency of DIT values on an identical basis for the EasyWay system. The EasyWay system shows a different trend to that of HSQLDB and JasperReports. After version 2, there is a drop in the number of classes at DIT level one of the inheritance hierarchy and then the DIT fluctuates until version 8. It then rises slowly until version 16, when the trend is then downwards again. Overall however, the net number of classes added at DIT levels one and two from a total of 14 classes added over all versions is 13 (i.e., 92.86%) of which 9 are at DIT level one. It is noteworthy that, in keeping with the result for the JasperReports system, there is also very little activity at DIT levels three and four for system 3; only one class is added in total to level three throughout – zero classes were added for DIT level four, which remained consistently at 1 throughout.

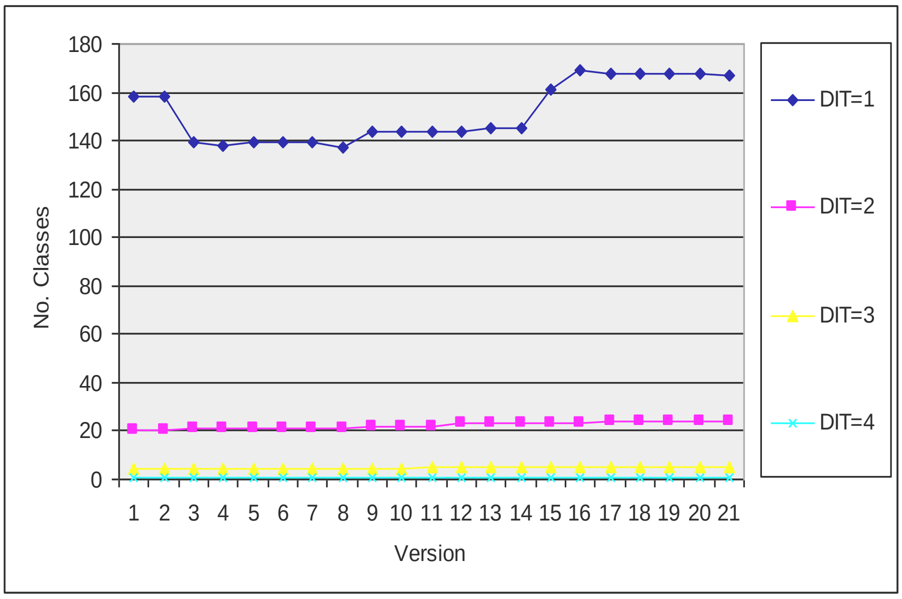


Figure 3.3. DIT frequencies EasyWay (all versions)

Figure 3.4 shows the DIT frequencies for the SwingWT system. Since the changes at DIT five, six and seven is not clear from figure 3.4, we show the fluctuation in classes at those levels in Figure 3.5. A clear trend for classes to be added at DIT level one is evident again. In fact, for DIT levels one and two, 400 and 83 classes were added, respectively. This compares with 19 added classes at DIT level three; a combined total of only 68 classes were added at levels four, five, six and seven. An interesting feature of levels five, six and seven is the fluctuation in the number of classes. Figure 3.5 illustrates this feature; while fluctuating, the trend for classes at DIT level five (and to a certain extent level six) is upwards.

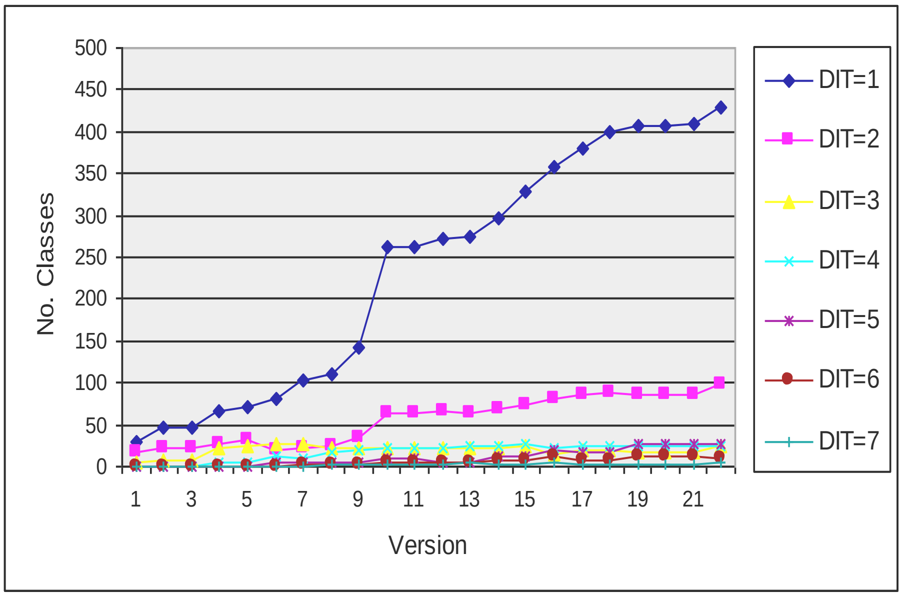


Figure 3.4. DIT frequencies SwingWT (all versions)

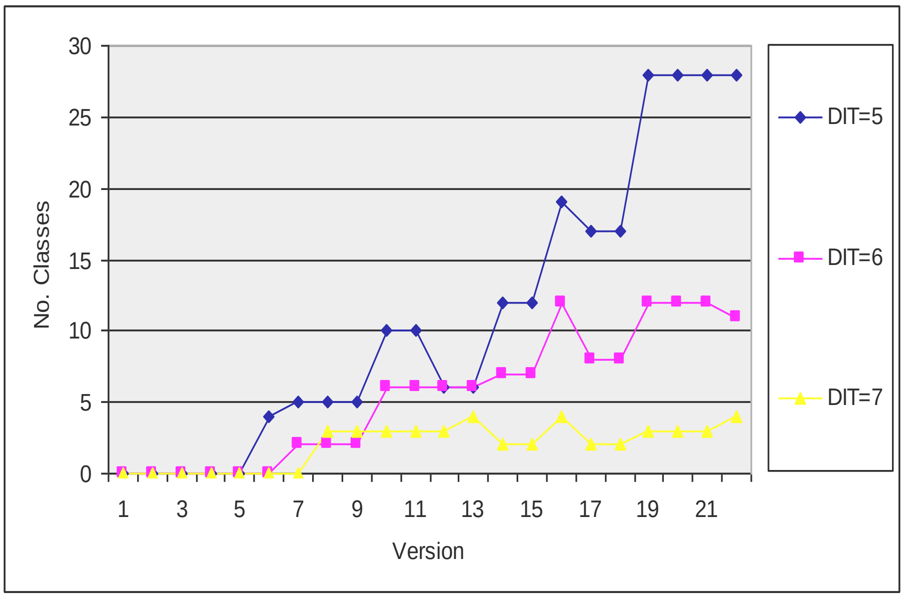


Figure 3.5. Classes at levels 5, 6 and 7 in SwingWT (all versions)

Figure 3.6 shows the trend in DIT frequencies for the JAG system. In contrast to data from the other four systems (with the possible exception of the EasyWay system), the DIT level one values remain relatively static over the course of the versions studied. Only 2 classes are added to level one in total between versions 1 and 23. The number of classes at level two actually falls from 15 to 12 over the same number of versions. For DIT levels three and four, in common with the JasperReports system, there is no change from their initial values.

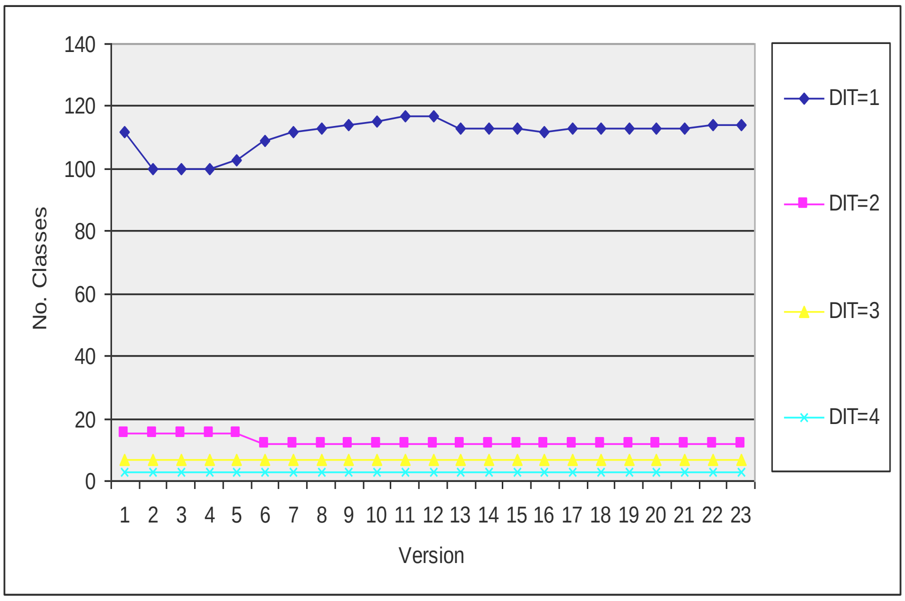


Figure 3.6. DIT frequencies JAG (all versions)

For scaling purposes, Figure 3.7 shows the DIT level one trend for the JBoss (the system has the highest number of start and end set of classes). A fluctuating pattern can be seen and the sharp peak seems to occur between versions 27 and 30. Figure 3.8 shows the DIT frequencies for the remaining DIT levels two to seven. A striking feature of Figure 3.8 when compared with Figure 3.7 is the strong similarity between the graphs for classes at DIT level one and those at DIT level two. Both graphs peak and trough at the same times and there seems a common symmetry between the two lines. There is also a noticeable correspondence (although not nearly as pronounced) between the line graphs for DIT level two and DIT level three. Both of these observations were unexpected results from the analysis; they suggest that there is a strong correlation between the numbers of classes found at DIT level one, DIT level two and, from the evidence presented, that at level three.

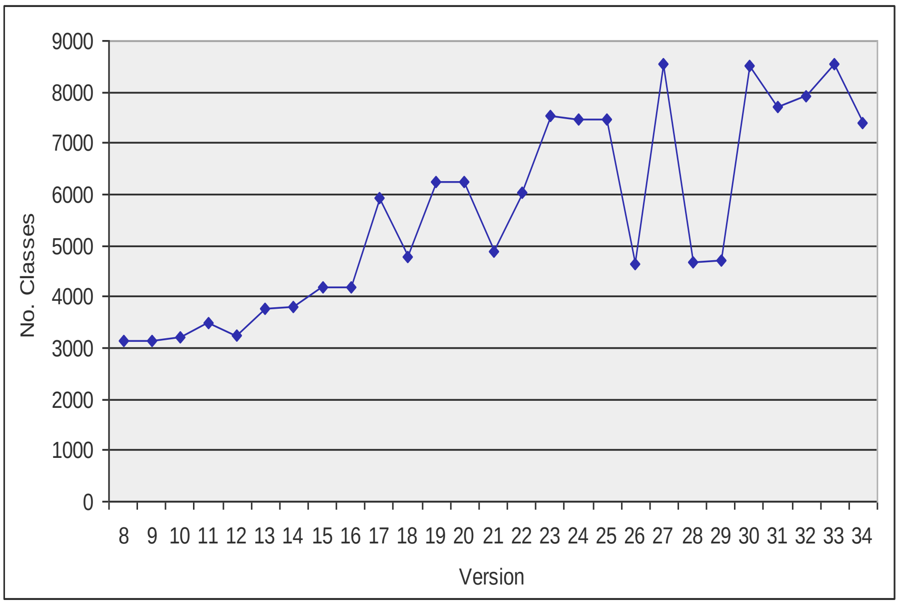


Figure 3.7. DIT level 1 frequencies for JBoss (all versions)

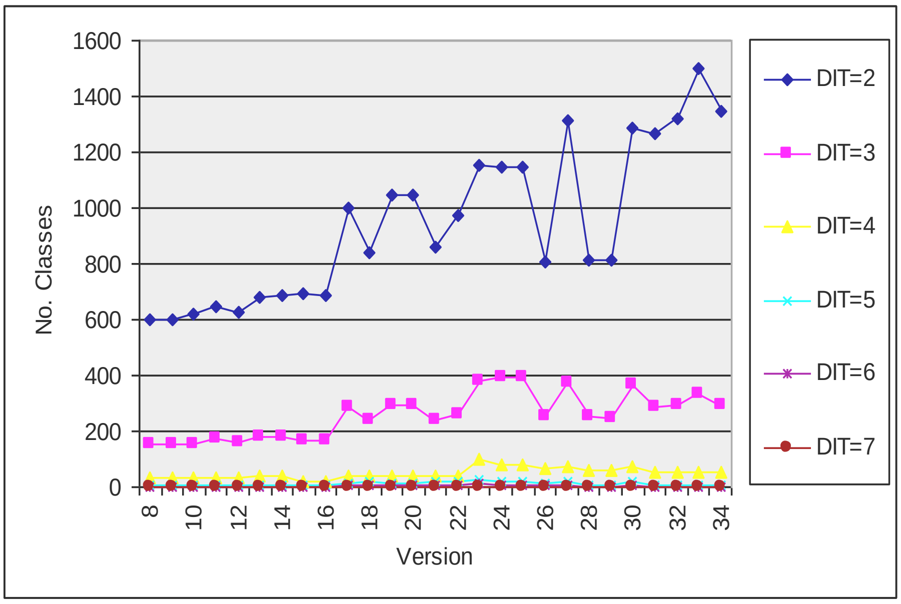


Figure 3.8. DIT level 2-7 frequencies for JBoss (all versions)

Figure 3.9 shows the trend in DIT frequencies for classes in the Tyrant system. Version 5 seems to be the point where significant changes are made to the classes at each level and the rise in DIT levels one and three values seems to be accompanied by a corresponding drop in DIT level two values. One noticeable feature of Figure 3.9 is the transition at version 26, when the number of classes at levels one, two and three move from a 'plateau-like' pattern and start increasing.

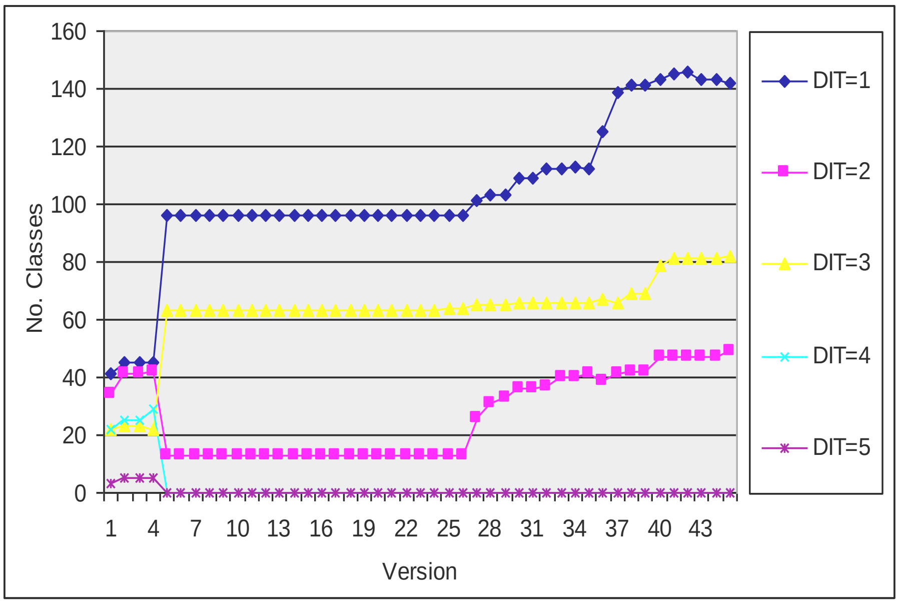


Figure 3.9. DIT frequencies for Tyrant (all versions)

The emerging theme from Figures 3.1-3.9 is clear in terms of where the majority of classes are added. For each of the seven systems analyzed, DIT level one is where the main activity lies. To emphasize the difference between DIT levels one, two and three we calculated that, from a total number of 6397 net added classes over all versions of all systems:

- 5181 classes (i.e., 80.99%) were added to DIT level one,
- 972 classes (i.e., 15.19%) were added to DIT level two and,
- 244 classes (i.e., 3.81%) added to DIT level three.

Moreover, only 25 classes were added to level four and 27 classes to level five (we note that only 4 of the 7 systems actually had classes at level five). At deeper levels, there is strong evidence of classes being removed. At DIT level five, 30 classes were added in total; at level six, 11 classes were added and at DIT level seven, only 4 classes were added.

3.4.2 Specialization and reuse ratio

The main objective of the research in this chapter was to show that the Java inheritance hierarchy tends to grow in width rather than depth. Based on previous studies (Bieman and Zhao 1995, Daly et al 1996, Harrison et al. 2000), we believe that developers will add classes to low (shallow) levels of the inheritance hierarchy rather than extend existing classes. One measure that might further inform our analysis is the Specialization Ratio (SR) (Henderson-Sellers 1995), which measures the extent of subclassing. A low SR implies that classes will tend to 'cluster' around lower (shallow) levels of the inheritance hierarchy (i.e., DIT levels one and two). A high specialization ratio suggests a high degree of subclassing. A further indication of the lack of subclassing is given by the Reuse Ratio (RR) (Henderson-Sellers 1995). An RR value close to 1 implies that the inheritance hierarchy is *narrow* and an RR value close to zero implies that the inheritance hierarchy is shallow (Henderson-Sellers 1995). Table 3.2 shows the summary data for the SR and RR metrics for the seven systems.

System	Med. SR	Max. SR	Med. RR	Max. RR
HSQLDB	0	0	0	0.8
JasperReports	0	0	0	0.86
EasyWay	0	0	0	0
SwingWT	0	14	0	0.75
JAG	0	0.33	0	0.86
JBoss	0	68	0	0.86
Tyrant	0	0	0	0.67

Table 3.2. SR and RR summary data for the seven systems

Table 3.2 gives a good representation of the lack of subclassing across the seven systems. The median SR and RR values are zero for all systems across all versions. Moreover, the maximum and standard deviation values represent values from a very small sample of classes for which the SR and RR were computed. For example, for version 1 of the JBoss system, the SR values for only 8 of the 3934 classes were non-zero (i.e., 0.2%); equally, the RR for only 99 of the same 3934 classes was non-zero (i.e., 2.52%). For version 16, only 9 SR or RR values from the 5085 classes in that version were non-zero. For the 9082 classes in version 34, only 8 SR values and 118 RR values were non-zero. The same pattern applied to each of the other six systems. The very low values for the SR and RR values imply, by definition, that reuse through subclassing was very low in each of the seven systems and that the shape of the inheritance hierarchy was very shallow. Considering the large number of classes added at DIT levels one and two and documented in the preceding sections, this did not come as a surprise. However, this evidence does support the claim of the research that developers do not tend to add classes at deep levels of the inheritance hierarchy, but rather at shallow levels, itself causing a broadening of the entire hierarchy.

3.4.3 Number of children

A final indication of the structure of the inheritance hierarchy and how it may evolve is given by the Number of Children (NOC) metric. The metric measures the number of immediate subclasses for a class. To find support for our original claim, we would expect:

1. A relatively high proportion of the classes at DIT levels one and two to have a large number of children.
2. Classes at DIT levels three, four, five, six and seven to have a very low proportion of children.

To investigate this feature, we ranked all NOC values in descending order and determined the DIT values for the first 50 classes in the generated sequence; we did this for both the first and last versions of each system (N.b., the SwingWT system only contains 50 classes in its first version explaining why we chose the number 50 as a sample size). The extracted profile is given in Table 3.3. For example, for the HSQLDB system, when we ranked the top fifty NOC classes, 48 of the classes inspected (i.e., 96%) had a DIT of one and only 2 classes had a DIT of two. It can be seen that the vast majority of the classes are taken from DIT level one. In every case except for the first version of Tyrant, over 50% of the top 50 classes when ranked on NOC were drawn from DIT level one. In over half of the cases, this percentage exceeds 70% and, in five cases, equals or exceeds 80%.

HSQLDB	**DIT=1**	**2**	**3**	**4**	**5**	**6**
First version	48 (96%)	2	0	0	0	0
Last version	39 (78%)	11	0	0	0	0
JasperReports						
First version	30 (60%	16	3	1	0	0
Last version	28 (56%)	18	3	1	0	0
EasyWay						
First version	43 (86%)	6	1	0	0	0
Last version	41 (82%)	8	1	0	0	0
SwingWT						
First version	29 (58%	16	5	0	0	0
Last version	28 (56%)	10	3	5	2	2
JAG						
First version	40 (80%)	5	5	0	0	0
Last version	40 (80%)	5	5	0	0	0
JBoss						
First version	37 (74%)	10	3	0	0	0
Last version	39 (78%)	9	2	0	0	0
Tyrant						
First version	16 (32%)	16	8	10	0	0
Last version	31 (62%)	11	8	0	0	0

Table 3.3. Breakdown of DIT ranked on NOC for first and last versions

Moreover, the top *ten* classes (ranked on NOC) were invariably drawn from DIT levels one and two. For example, of the top ten classes for the first version of HSQLDB, 9 were at DIT level one and 1 at DIT level two. Equally, for the first version of SwingWT, the top ten classes comprised 8 classes at DIT level one and 2 classes at level two. For the final version of the JBoss system, 9 of the top ten classes were at DIT level one and only 1 at DIT level two. Figure 3.10 shows this trend and the large number of children associated with those classes (NOC values actually ranged from 14 to 69); this breakdown is typical of the seven systems studied. Figure 3.11 shows the same data for the final version of Tyrant.

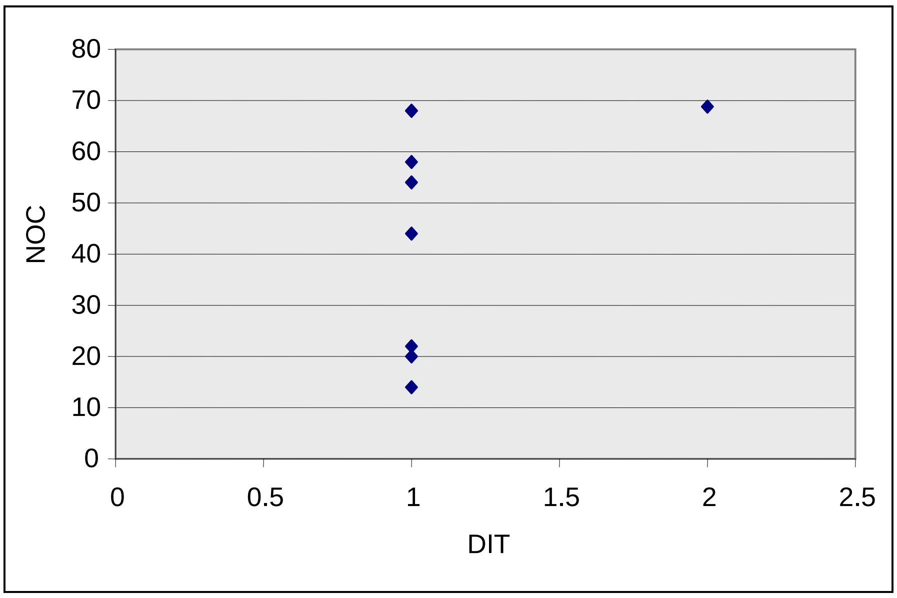


Figure 3.10. DIT and ranked NOC for JBoss

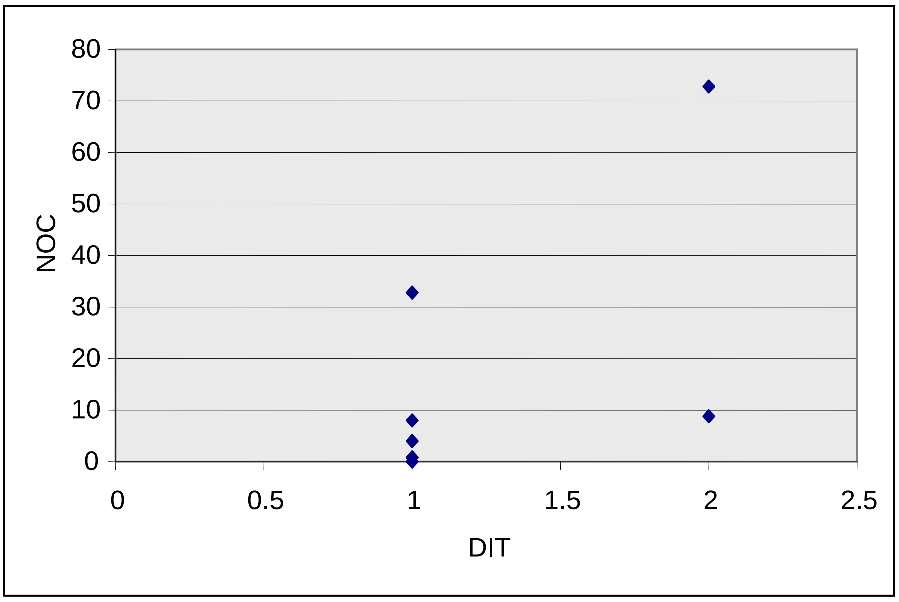


Figure 3.11. DIT and ranked NOC for Tyrant

The data in Table 3.3, and the evidence presented confirms our claim that the majority of activity is at DIT levels one and two, with very little activity at, and beyond, level three. Only 21 of the 700 classes (i.e., 3%) from Table 3.3 were found to be at levels five to seven.

3.4.4 Coarse-grain Method and Attribute analysis

The preceding analysis shed some light on the trends of changes of inheritance at class level. In this section we empirically investigate the *evolutionary* behaviour of inheritance at a finer-grain (at method and attribute level) in a subset of the systems (EasyWay, JAG and JBoss, see systems 4, 6 and 7 in Section 2.3.2.5). The purpose of this analysis is to determine if patterns in methods and attributes follow the same trend as that of classes.

3.4.4.1 Method Analysis

Table 3.4 presents the summary data for the three systems in the form of maximum (Max.), minimum (Min), median (Med) and Mean change values in the number of methods across the versions of the three systems (EasyWay, JAG and JBoss). The 'Ch' in Table 3.4. indicates an increase or decrease in the number of methods (NOM). In keeping with the format of Table 3.1, Table 3.4 also shows the normalized (Norm.) percentage of the Max. to indicate what percentage of initial system size (in NOM) that Max change represents. For example, the Max. change of 220 for the EasyWay represents an increase of 17% in that system over its original size (1266 methods). Similarly, Table 3.4 also shows the variance (Var.) values for the set of changes (in NOM) in versions of the systems. For example, the variance of the set of changes from version to version of EasyWay was 3893.

System	Max.Ch	Norm.	Min.Ch	Var.	Med.Ch	Mean Ch
EasyWay	220	17%	0	3893	4	30
JAG	100	8%	0	655	8	19.14
JBoss	26934	96%	24	76849781	5366	8046.80

Table 3.4. Summary method change data for the three systems (all versions)

From Table 3.4, we see that the mean values are always positive, suggesting the growth of the systems in terms of NOM. The low values of variance and median change for EasyWay and JAG suggest that the systems are the most stable in the three systems studied. The normalized percentage (Norm.) change value for JBoss indicates that the system changes significantly in terms of NOM from its initial versions. Considering the Norm. change values we see that JAG, seems to be the most stable system in terms of NOM. This can also be seen in Table 3.1 where the Norm. value for JAG was 2%.

Figure 3.12 shows the frequencies of NOM at each DIT level for the EasyWay system. From Figure 3.12 the NOM at DIT one falls after version 2 and then fluctuates until version 14 which then starts to rise in version 15. The total NOM added at DIT one and two is 184 (i.e., 96.84%) of which 141 methods were added at DIT one and 43 methods added at DIT two. The total NOM added at DIT three was 6 (with the addition of only 1 class, see Figure 3.3) which accounts for 3.16% overall; the NOM at DIT four stayed static throughout the entire set of 21 versions. The change in NOM for EasyWay shows a similar pattern to that of changes in number of classes (see Figure 3.3).

At a lower granularity we note that lower level (method and attribute level) analysis can often show the changes that may go undetected when analyzing the system at a higher granularity (i.e., class and package level). For example, in the transition between versions 2 to 3 of EasyWay, the number of classes at DIT two increases by only 1 class (from 20 to 21); however, in the same transition the NOM in the same level increases by 35 methods. Similarly, between versions 4 to 5, the number of classes at DIT two stays constant; however, the NOM in the same level increases from 190 to 191. This suggests that while a system may not change in number of classes, there may be significant *within-class* maintenance activity in the system.

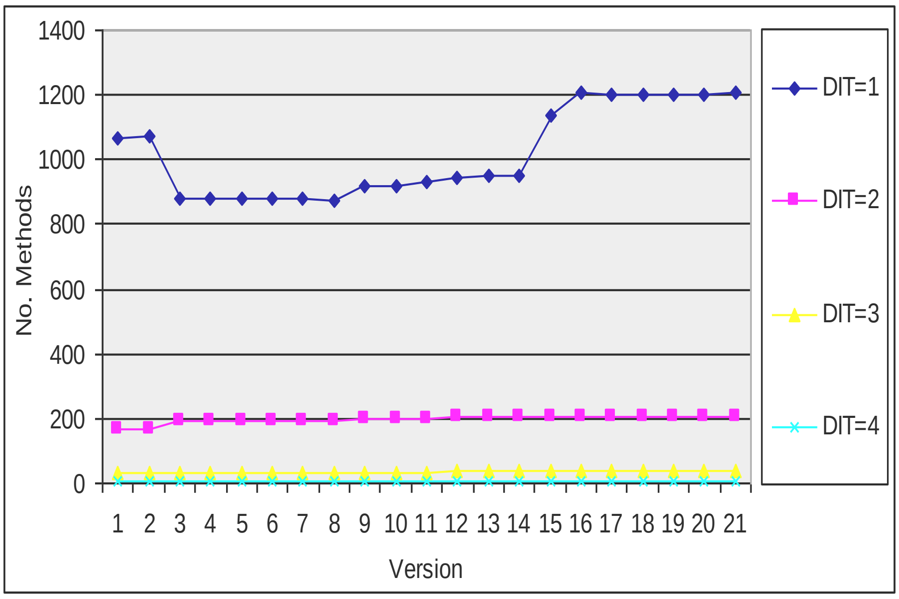


Figure 3.12. NOM frequencies for EasyWay (all versions)

Figure 3.13 shows the frequencies of NOM at each DIT level for JAG system. From Figure 3.13, a strong tendency can be seen for methods to be added at DIT one. A total of 290 methods were added at DIT level one accounting for 100% of all methods added. The NOM at DIT level two started at 99 which then fell to 69 in version 6 and remained static throughout the 23 versions of the system. Interestingly, the NOM at DIT three and four also remained static at 67 and 3 respectively, throughout the remaining versions of the system.

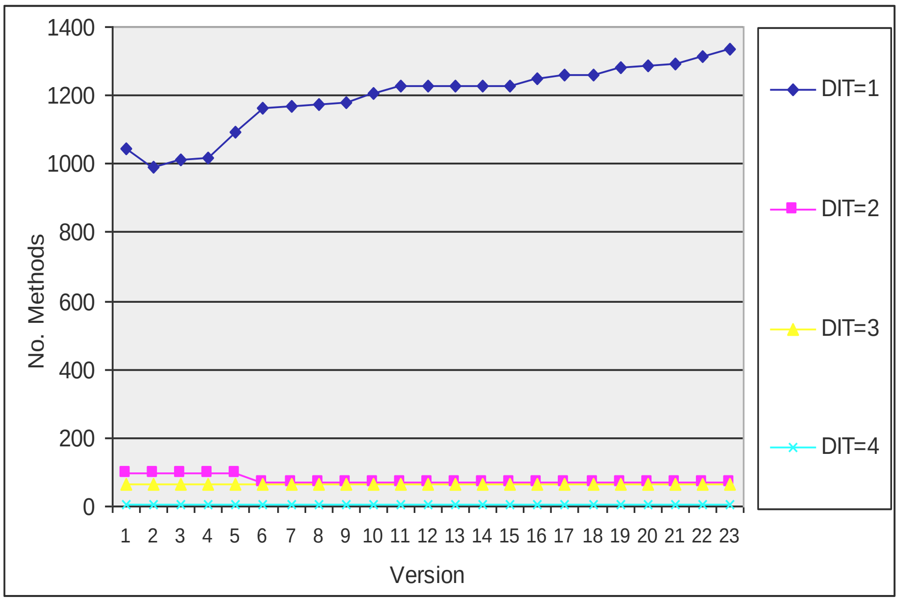


Figure 3.13. NOM frequencies for JAG (all versions)

Figure 3.14 shows the frequencies of NOM at each DIT level for JBoss system. For succinctness, Figure 3.14 only shows the breakdown of the NOM at DIT one. JBoss is the largest system in number of classes, methods and attributes. A fluctuating trend can be seen in NOM at DIT one similar to that observed in number of classes (Figure 3.7). Figure 3.15 shows the frequencies of NOM at DIT two to seven in JBoss. In Figures 3.14 and 3.15, the NOM at DIT one and two tends to change at the same time. This may be due to the fact that the number of classes at these two DIT levels (see Figures 3.7 and 3.8) tends to change in the same manner as the NOM in Figures 3.14 and 3.15. In JBoss, the total number of methods added was 33705 of which 28420 (i.e., 84.32%) were added at DIT one, 4107 (i.e., 12.18%) added at DIT two and only 1178 (i.e., 3.49%) added at DIT levels three to seven.

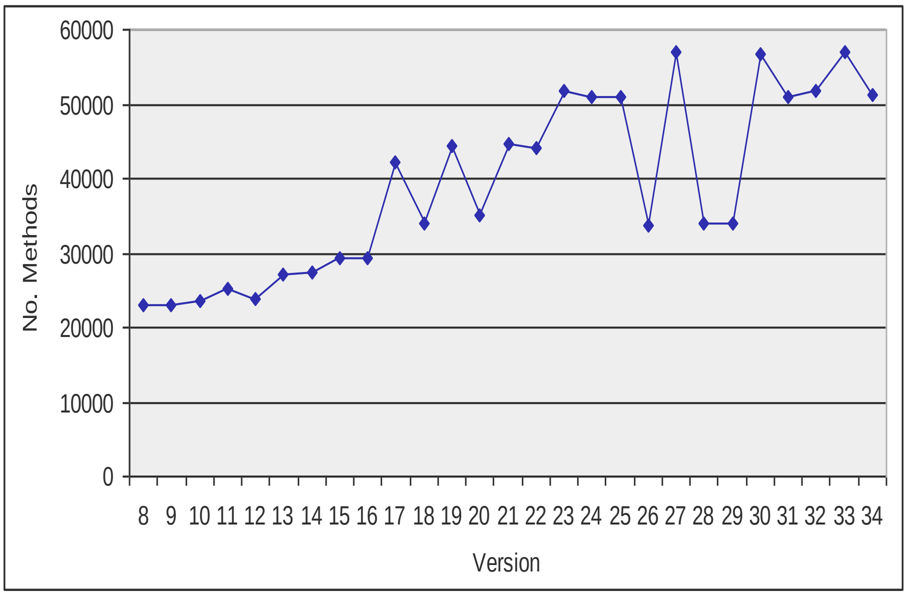


Figure 3.14. NOM frequencies for JBoss for DIT 1 (all versions)

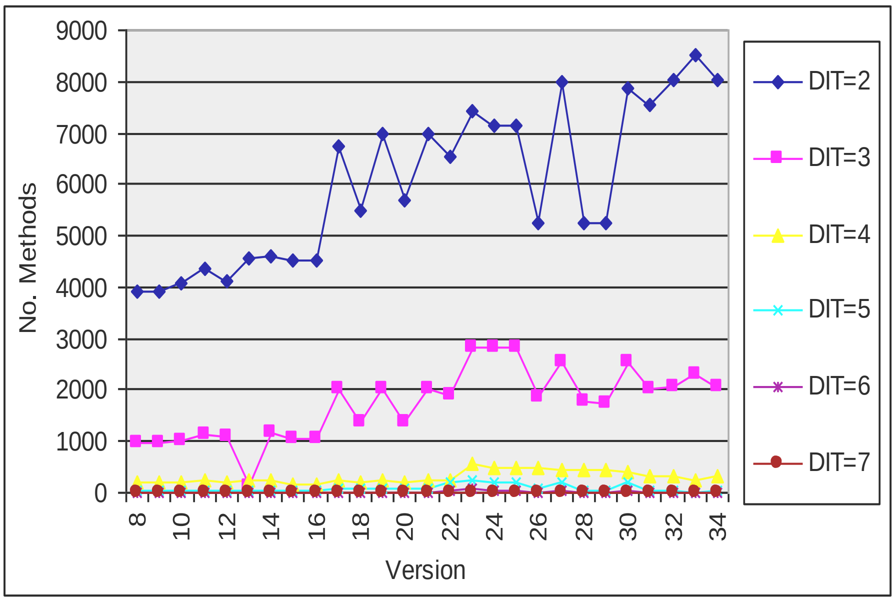


Figure 3.15. NOM frequencies for JBoss for DIT 2-7 (all versions)

To summarise the analysis of NOM, we calculated the changes in terms of increase (additions) and decrease (deletion) of methods in all versions of the three systems. We found that, from a total of 210238 changes (additions and deletion of methods) in all versions of the three systems:

1. 173932 changes (i.e., 82.73%) were made to classes at DIT level one,
2. 22428 changes (i.e., 10.67%) were made to classes at DIT level two and,
3. 13878 changes (i.e., 6.6%) were made to classes at DIT level three and beyond.

3.4.4.2 Attribute Analysis

In addition to classes and methods, we also analyzed the changes in the systems from an attribute perspective in the three systems. Table 3.5 shows the summary data for changes of number of attributes (NOA) in the three systems, in order of number of versions studied in the same format as Tables 3.1 and 3.4.

System	Max. Ch	Norm.	Min. Ch	Var.	Med. Ch	Mean Ch
EasyWay	136	13%	0	1687	1	18.8
JAG	83	14%	0	494	11	17.72
JBoss	10327	98%	7	11271927	1535.5	2888.38

Table 3.5. Summary attribute change data for the three systems (all versions)

From Table 3.5 we observe that the values of variance and median change of NOA for EasyWay and JAG were relatively low. The two systems were therefore considered as the most stable systems (in terms of NOA). Interestingly, these two systems were also considered as the most stable systems in terms of number of classes and methods studied. The variance value for JBoss is considerably higher in comparison with the other two systems. For scaling purposes, Figure 3.16 shows the total changes of NOA at each DIT level for EasyWay and JAG systems and Figure 3.17 shows the same trend for JBoss.

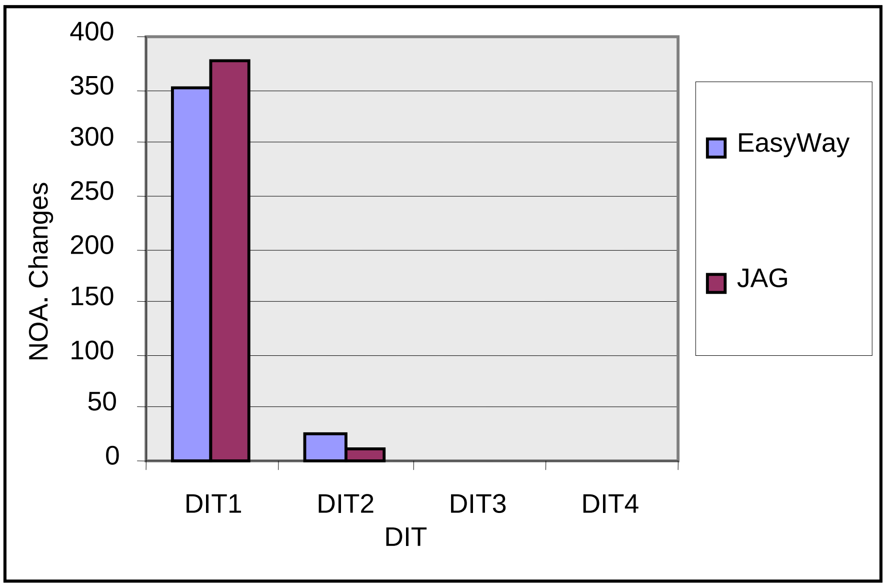


Figure 3.16. The total changes of NOA for Easy and JAG (all versions)

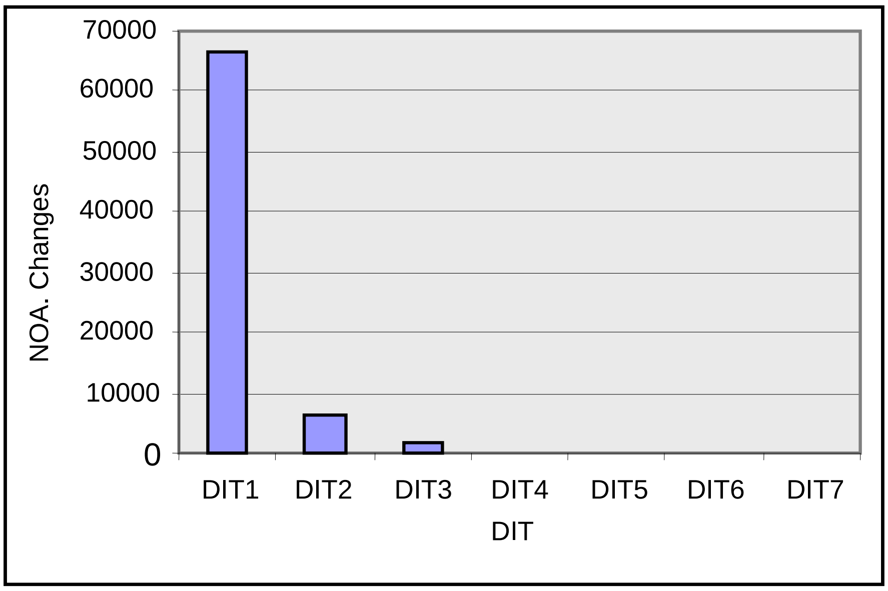


Figure 3.17. The total changes of NOA in JBoss (all versions)

From Figures 3.16 and 3.17, we see that the vast majority of the changes of NOA occur at DIT one in the three systems studied and the trend of changes is declining as the DIT increases. This clearly indicates that changes are not spread proportionately.

To summarise the analysis of NOA, we calculated the total and percentage of changes of NOA at DIT levels one, two and beyond in all versions of the three systems studied. We observed that, from a total of 75864 net changes over all versions of all systems:

1. 67054 (i.e., 88.39%) changes of attributes were made to classes at DIT one,
2. 6352 (i.e., 8.37%) changes of attributes were made to classes at DIT two and,
3. 2458 (i.e., 3.24%) changes of attributes were made to classes at DIT three and beyond.

3.5 Discussion

Many issues arise from the analysis in this chapter. The population was a non-trivial set of OSS projects that had undergone protracted maintenance. One important aspect that needs to be considered is the threats to the validity of the study. Firstly, we have to consider the extent to which our non-random sample has impacted our ability to generalize. We have chosen a set of application domains ranging from computer games to a database application. Secondly, we have looked at different numbers of versions of each of the seven systems. While ideally, we would have liked to have had the same number of versions for each system, we wanted to extract as much information about available data as possible. Thirdly, while we can make observations about numbers of classes, methods and attributes at different levels of the inheritance hierarchies, we can not say with any certainty, or quantify with any certainty, the movement of classes between different levels. This finer-grained analysis of class movement and relocation in four systems is described in chapter 5. Fourthly, since we restrict our analysis to structural aspects of the evolution we do not know *why* the developers made the choices that they have. A question that arises from the study is whether we should consider the evolution of systems at shallow levels as bad practice, since it contradicts the original aim of inheritance? Our belief is maybe not. Developers will nearly always modify systems in the easiest and quickest way possible and from that perspective we could not really expect ‘ideal’ trends to occur. Furthermore, systems will inevitably deteriorate over time and re-engineering effort by developers is a luxury that cannot usually be afforded. In other words, it is not bad practice that leads to evolution at shallow levels, merely a ‘fact of life’ in the maintenance world that systems will evolve in a manner that conforms to forces dictated by the original architecture and by previous maintenance effort. Many systems may not be amenable to deep inheritance hierarchies in the first place, so any additional classes will always be placed at shallow levels. Previous studies have suggested that graphical-based systems are the most amenable to extension through inheritance (Harrison et al. 2000). Interestingly, the SwingWT system in our study did exhibit high levels of inheritance up to DIT seven.

One interesting aspect of OSS is that the developers are often geographically and often time-zone separated from each other. Often the design documents are not available to each of the 'contributors'. We offer the explanation that for OSS, developers may add classes at shallow levels of the inheritance hierarchy because they are unaware of the 'bigger design picture'. Of course, this does not explain why for previous studies where proprietary software was used, the same observations have been made, although scale might have a similar impact. In addition, an anecdotal claim of many developers is that the original designs of many proprietary systems are not updated as and when changes to the software are made and this renders those designs virtually unusable. The explanation for the lack of available design documentation in OSS may therefore be mirrored by outdated designs in proprietary software.

We also need to consider the implications of our study. One major implication of the effective *flattening* of the inheritance hierarchy is the potential maintenance headache of modifying a class with many children (i.e. its dependencies). Inheritance is a form of coupling (Briand et al. 1999b) and, in this sense, a short-term 'easy fix' may be at the expense of long-term problems - refactoring may have a large role to play in this sphere of developer activity (Fowler 1999). Finally, it is interesting and ironic that there is previous empirical evidence to suggest that deep levels of inheritance have been blamed for the existence of faults (Cartwright and Shepperd 2000); yet, we could suggest that by avoiding those deep levels of inheritance, the problem may simply have been devolved to shallower levels of inheritance (further empirical studies would be needed to support this claim).

3.6 Summary

In this chapter, we have described an empirical analysis of the trends in inheritance over multiple versions of Java OSS. Previous studies have suggested that developers tend to avoid the use of inheritance at deep levels (Cartwright and Shepperd 2000, Bieman and Zhao 1995) and that consequently, systems will evolve at very shallow levels (they will grow ‘breadth-wise’ rather than ‘depth-wise’). The aim of the research described in this chapter was to demonstrate whether or not this was the case. A tool (JHawk) was used to extract inheritance-based metrics from seven OSS. In addition, we extracted size based metrics (number of methods and attributes) from three systems to investigate the finer-grain changes of inheritance. The results confirm for the set of OSS studied what many of the earlier reported studies did for proprietary systems (i.e., low DIT levels). There is also a strong tendency for classes, methods and attributes to be added at levels one and two of the hierarchy rather than at deeper levels. Over 96% of classes added over the course of the versions of all systems were either at level one or level two of class hierarchy. This result was supported through analysis using the Specialization Ratio, Reuse Ratio and Number of Children metrics, which showed the extent of reuse in, the shallowness of, and width within, the inheritance hierarchy, respectively. These metrics supported and informed our analysis of the DIT and NOC metrics forming the main thrust of the investigation. Furthermore, we observed that approximately 93% of method and 97% of attribute changes were made to classes at DIT levels one and two in three of the seven systems studied. Only over 6% of method and 3% of attribute changes were made to classes at DIT three and beyond.

The results have relevance for developers in terms of systems maintenance and refactoring. Predicting change-prone areas of systems will help to target refactoring effort and this may impact the localization of faults. If the majority of additions of classes, methods and attributes are made at shallow levels of the hierarchy, then that is *possibly* where the faults will be likely to be found as a system evolves.

Since the focus of this Thesis is on trends in inheritance, the following chapter will explore the changes of methods and attributes at each DIT level and compare those changes to a set of low-level refactorings applied to initial versions of the systems. The refactoring data was originally extracted for a previous study (Counsell et al. 2006a) when only initial versions of the systems were available.

CHAPTER 4 Method and Attribute Evolution and their Refactorings

4.1 Introduction

In the previous chapter we described the evolution of inheritance by analysing net changes of inheritance at class level. In Chapter 3 we also empirically investigated the trends of inheritance at method and attribute level and compared the trends to that of class changes in a subset of the systems.

While we usually expect an OO system to grow (in number of classes) as it ages, what are not so obvious are patterns in the evolution of specific class features. In this chapter, we explore empirical traits of four Java OSS using data extracted by two tools and informed by a previous study of inheritance depth evolution (Nasseri et al. 2008) which suggested that a vast majority of changes were made at inheritance levels one and two. The implication of this result is that, other things remaining equal, the majority of faults will be invested where the majority of functionality is found (i.e. at levels one and two). In this chapter, in contrast to the previous, we analyse evolution at the lower level of granularity given by the 'methods' and 'attributes' of a class on an incremental (change per version) basis rather than absolute class size per version, as studied in the previous chapter.

Exploration of this system facet may inform better targeted re-engineering effort, since it ignores the overall pattern in favour of low-level, 'local' change. Equally, while it is valuable and interesting to study patterns of change in classes, the *addition* of methods and attributes to classes may not necessarily follow the same pattern. In addition, evolution at a finer-grain can identify trends not possible on a class-wide basis; the approach thus represents a 'white-box' view of the investigation of evolutionary forces. Our analysis also allows direct comparison with a set of low-level refactorings extracted by an automated tool for a previous study (Counsell et al. 2006a).

Section 4.2 describes the motivation for the empirical study. In Section 4.3 we outline the design of the empirical study including the four systems used, data collected, and summary data. We then present data analysis (Section 4.4). Section 4.5 provides a discussion of the results before we present an overall summary of our empirical study in Section 4.6.

We note that a part of this chapter (method evolution and overall refactorings applied) has been accepted (as one of the best papers) for publication in (Nasseri and Counsell 2009b) and the remaining part (attribute evolution and attribute related refactorings applied) has been accepted for publication in (Nasseri and Counsell 2009a).

4.2 Study Motivation

Systems will inevitably grow larger as they evolve even if significant re-engineering is applied to them at frequent intervals. In Chapter 3 we found that the vast majority of changes of inheritance occurred at levels one and two of inheritance hierarchy. Very little activity was found at level three and beyond. The motivation for our empirical study in this chapter stems from two sources. Firstly, there is a clear distinction between maintenance of a system through regular changes and that associated with refactoring. The latter represent 'semantic preservation' re-engineering of the code while the former can represent almost any other change made to code. In this chapter, as well as investigating the trends in method and attribute addition through versions of OSS, we are also interested in exploring whether low level refactorings (within-class refactorings) are undertaken independently by developers on a widespread scale or it is simply the case that refactoring tends to focus on manipulation of methods and attributes and classes at the same time. Secondly, in theory, an added class should make use of available methods and attributes from the classes it *inherits* from, thus conforming to class hierarchy specialisation. But we pose the question as to whether this actually happens in reality. Scrutiny of trends in methods and attributes was further motivated by the fact that the vast majority of refactorings apply not at the class level but at the method and attribute level.

Moreover, many of the refactorings that developers are likely to undertake are at the method and attribute level (moving methods/fields, extracting methods and renaming methods/fields pulling up/down methods/fields being examples (Fowler 1999)); it therefore makes sense to explore whether evolution at that level can give a developer any insights not afforded at the higher class level. The implication of the analysis and results is that maintainers should not necessarily look at high-level, class-based system trends when considering re-engineering effort, but on the incremental low-level features of a system. As a *prima facie* guide to where, first, remedial effort may need to be applied and, second, to provide an insight into where refactoring effort should be applied, analysis at the method and attribute level seems to show significant promise.

The contribution of the empirical study in this chapter is that the authors know of no other studies that have looked longitudinally and specifically at the evolution of class methods and attributes and the relationship with their enclosing classes and that of refactoring. The over-riding message that the chapter presents is that empirical studies targeted purely at the class level may miss deviant behaviour at lower levels of granularity and the opportunity for remedial action therein.

4.3 Study Design

4.3.1 The four open-source systems

We base our analysis on evolution of methods and attributes of four Java OSS (the same investigation for three systems in our system archive is presented in Chapter 3). The systems used are as follows: HSQLDB, JasperReports, SwingWT and Tyrant (see systems 1, 3, 5 and 8 in Section 2.3.2.5). For this study, we included all available versions of the systems.

4.3.2 Data Collected

We again used the JHawk tool (described in Section 2.3.2.6) to collect inheritance and size metrics from each version of the four systems. The metrics collected were as follows: Depth of Inheritance Tree (DIT), Number Of Methods (NOM), and Number Of Attributes (NOA) (see metrics 1, 5 and 6 in Section 2.3.2.6). In addition, we used the refactoring data extracted from initial versions of the three systems (HSQLDB, JasperReports and Tyrant; when the refactoring tool was run only initial versions of the systems were available) for a previous study (Counsell et al. 2006a). In (Counsell et al. 2006a) the SwingWT system was not included and the refactoring data for SwingWT is therefore not presented in this chapter either.

4.3.3 Summary Data

Table 4.1 presents the summary data for the four systems with maximum (Max.), minimum (Min), median (Med) and mean change (Ch) values in NOM across the versions of the four systems. Table 4.1 also shows the normalized (Norm.) percentage of ‘Max.’ indicating what percentage of initial system size (in NOM) ‘Max.’ represents. For example, the maximum change of 2073 for HSQLDB represented an increase of 213% in that system over its initial size; finally, Table 4.1 also shows the variance (Var.) for the set of changes in versions of the systems.

System	Max. Ch	Norm.	Min Ch	Var.	Med Ch	Mean Ch
HSQLDB	2073	213%	15	667782	684	777.0
JasperReports	838	10%	29	58851	164	238.09
SwingWT	1929	510%	45	232104	370	519
Tyrant	617	63%	0	10232	10.5	47.70

Table 4.1. Method change data for the four systems

The normalized percentage (Norm.) change values for SwingWT and HSQLDB indicate that these two systems changed significantly in terms of NOM from their initial versions. The most striking feature of Table 4.1 is the low median change value for Tyrant. This was a surprising result considering the fact that the system evolved to its 45th version and we expected the system to be the most changeable system in the set of systems studied. Furthermore, the high mean change value for HSQLDB indicates that significant change (in NOM) has been applied to the system. Table 4.2 presents the summary data in the same format as Table 4.1 for the changes of NOA in the four systems.

System	**Max Ch**	**Norm.**	**Min Ch**	**Var**	**Med Ch**	**Mean Ch**
HSQLDB	2028	326%	1	682104	387	583.2
JasperReports	479	13%	7	18179	109	144.72
SwingWT	645	244%	20	25929	165	206.8
Tyrant	341	26%	0	5037	1.5	31.48

Table 4.2. Attribute change data for the four systems

From Table 4.2, the high normalized percentage (Norm.) values for HSQLDB and SwingWT compared with the other two systems indicate that these two systems again grew significantly in terms of number of attributes from their initial versions. This trend was also found in Table 4.1. The JasperReports and Tyrant systems, on the other hand, showed relatively slower growth over the period investigated, suggesting that net addition of attributes was far less frequent.

4.4 Data analysis

The maximum DIT of any class in HSQLDB system was 4. Figure 4.1 shows the frequencies of the NOM in the HSQLDB system across its six versions. From Figure 4.1, classes at DIT level one have the highest growth rate in terms of NOM. Classes at DIT level two and level three also show increases, but at a slower rate. Overall, 3855 methods were added over the course of the six versions, of which 2928 (i.e., 75.95%) methods were added at DIT level one and 756 (i.e., 19.61%) at level two. Only 171 (i.e., 4.43%) methods were added to DIT level three over the course of the six versions studied.

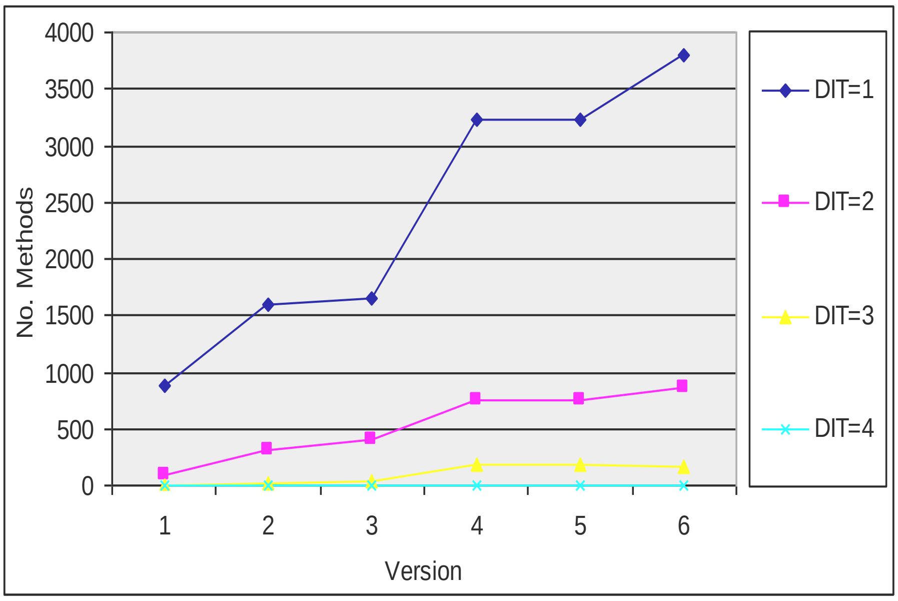


Figure 4.1. NOM frequencies HSQLDB (all versions)

Figure 4.2 shows the net NOM added or removed from the versions of HSQLDB on an incremental basis. For example, between versions 1 and 2, 1517 methods were added at DIT level one, 253 methods added at level two and 137 methods added at level three. Equally, the net added NOM falls between version 2 and 3 at every level. A clear single 'peak' effect is evident from the figure, suggesting that significant effort was applied to DIT levels one and two between versions 3 and 4. Between versions 4 and 5, more methods were removed than added to level one (hence the negative value). As noteworthy is the fact that HSQLDB saw only a relatively small rise in its overall number of classes from version 2 to 3 (from 130 to 147). Had we used just classes as a basis of our analysis as in the previous chapter, we may therefore have missed an important trend in the evolution of this particular system.

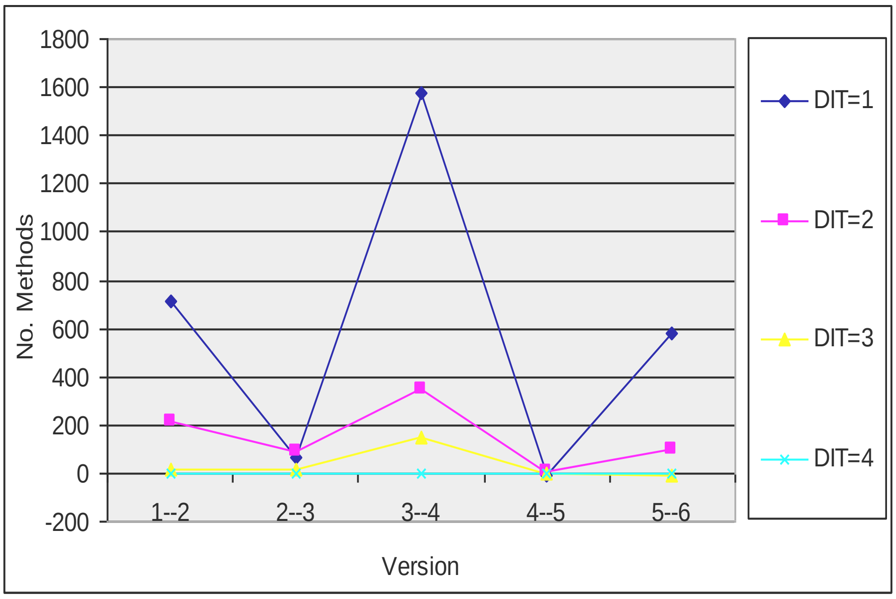


Figure 4.2. Net changes in NOM HSQLDB (all versions)

One feature of the evolution of a system that may generally help to explain the trends in Figure 4.2 is that of refactoring, since according to Fowler (Fowler 1999), evolutionary ‘decay’ is impeded by consistent application of refactoring techniques. Figure 4.3 shows the trend in refactorings applied to this system in the first four versions. Refactoring data was extracted using an automated tool, details of which were first reported in (Advani et al. 2006). The tool, capable of extracting fifteen different types of refactoring, was run against each version of the source code of a particular system and the results tabulated.

From Figure 4.3, it is noticeable that the single ‘peak’ of refactorings (between versions 3 and 4) occurred at the same time as the single ‘peak’ of net additions of methods to the HSQLDB system shown in Figure 4.2. This suggests that refactoring effort was applied to this system at the same time as large-scale addition of methods. Inspection of the actual refactorings undertaken from the data available revealed a high percentage of the ‘Move Method’, ‘Rename Method’ and ‘Move Field’ refactorings (Fowler 1999). As their names suggest, all three refactorings are directly related to the movement of methods and attributes around classes. (We could *not* have extrapolated that information through analysis of class evolution alone).

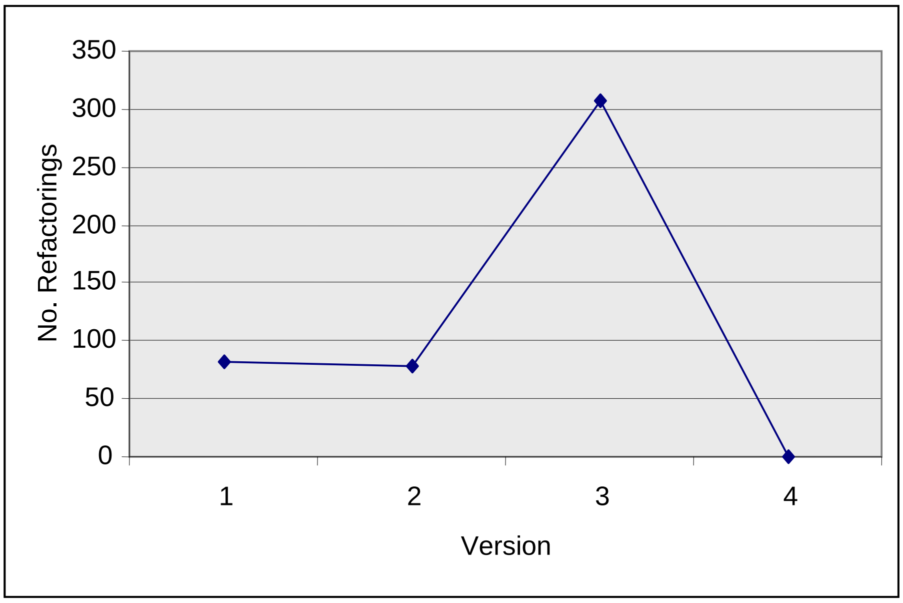


Figure 4.3. Refactorings in HSQLDB (4 versions)

Further analysis of HSQLDB revealed that most classes were added between versions 3 and 4 (176 classes overall). Figure 4.4 shows the frequencies of NOA in the HSQLDB system across its six versions. From Figure 4.4, a strong tendency can again be seen for attributes to be added into classes at DIT level one. The number of attributes at DIT levels two and three also increases, but to a lesser extent. Overall, 2926 attributes were added across six versions of the system, of which 2611 (i.e., 89.23%) were added at DIT level one, 295 (i.e., 10.08%) added at level two and only 20 (i.e., 0.68%) added at level three. The number of attributes at DIT level four showed no increase whatsoever over the six versions of the system. One pertinent question that arises from this initial analysis is whether any classes were added at DIT level four throughout any of the versions? Inspection of the data revealed that only one class was added at that level throughout the six versions investigated (i.e., between versions 3 and 4; that class was removed in a later version (5-6)).

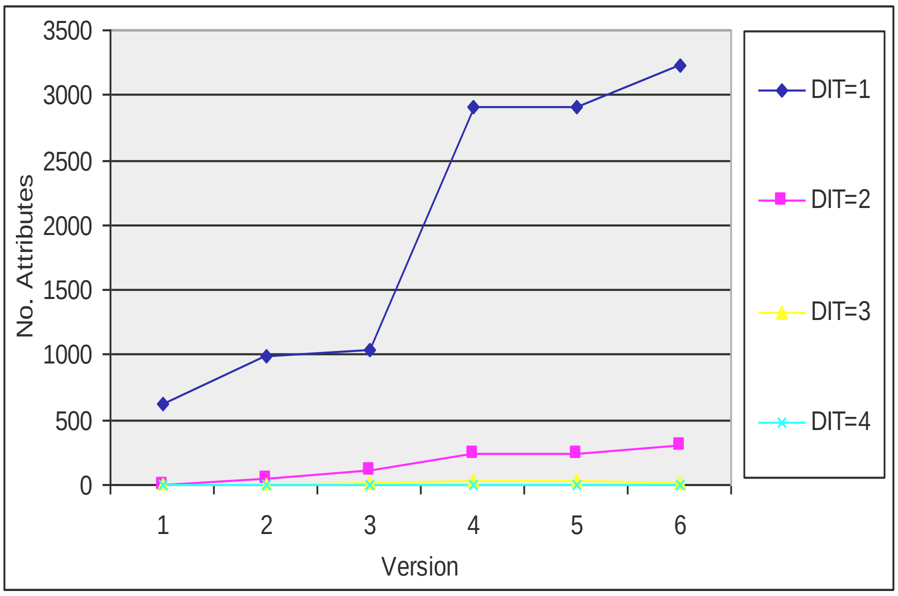


Figure 4.4. NOA frequencies HSQLDB (all versions)

Figure 4.5 shows the *net* NOA added or removed (net changes) from the versions of HSQLDB on an incremental basis. For example, from Figure 4.5, the net NOA added at DIT one between version 1 and 2 was 377; 46 attributes were added at DIT two and only 4 attributes added at DIT three. It is notable that while 12 classes were added at DIT level three throughout the versions studied, only 4 attributes were added in that time. From the same figure, the maximum change of NOA takes place between versions 3 to 4. This trend can also be observed in changes of NOA (Figure 4.2) suggesting that the system underwent major re-engineering between these two versions. From Figures 4.4 and 4.5, we see that DIT one and two is where the vast majority of activity (addition and deletion of NOA) takes place; developers tended to be relatively inactive at deeper levels of the inheritance hierarchy.

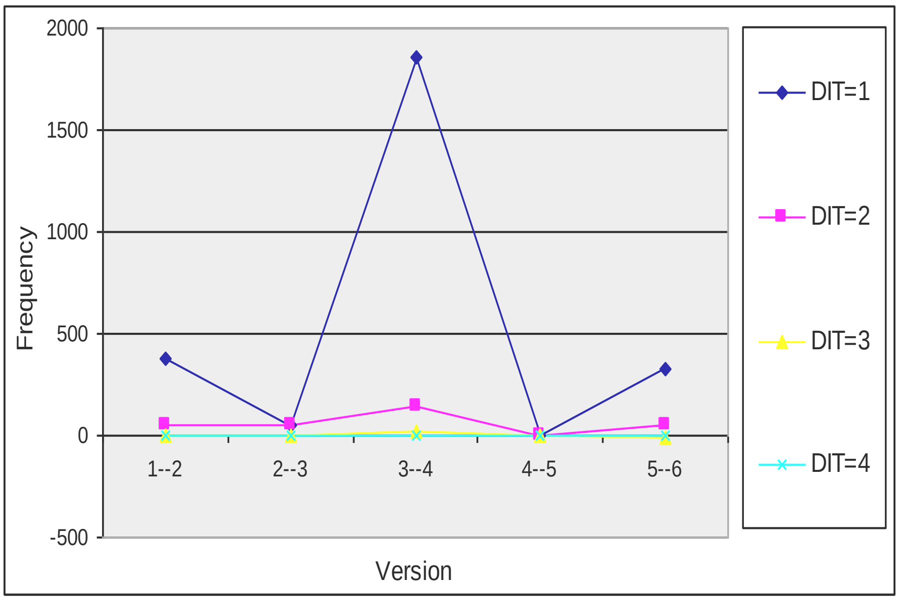


Figure 4.5. Net changes in NOA HSQLDB (all versions)

The actual values of the net changes of classes at different DIT levels in six versions of HSQLDB are summarized in Table 4.3. There is a clear trend for classes to be added at shallow levels of the hierarchy and not necessarily in version 1, but between versions 3 and 4. If classes are being added primarily at levels one and two, then we could probably expect a corresponding rise in added attributes for those classes. This is based on the fact that classes at levels one and two by sheer virtue of their level would have, on average, fewer inherited class features than let's say a class at level four. By definition, the deeper the class, the greater opportunity for inheritance from classes above. Figure 4.5 and Table 4.3 show that net change in NOA is not always accompanied by a corresponding change in classes.

Version	DIT1	DIT2	DIT3	DIT4	Total
1-2	50	21	3	0	74
2-3	10	6	1	0	17
3-4	133	34	8	1	176
4-5	-1	1	0	0	0
5-6	33	4	-1	-1	35

Table 4.3. Net class additions HSQLDB (All versions)

According to Fowler (Fowler 1999) developers should refactor 'mercilessly' and apply various types of refactoring as good practice. However, all empirical evidence to date suggests that only simple refactorings are undertaken frequently. For example, two studies by Counsell et al. (Counsell et al. 2003, Counsell et al. 2006a) have found that the majority of refactorings were simple renaming of methods and fields. More 'complex' refactorings such as those related to inheritance were found to be applied less frequently. One could argue that renaming and moving features is part of any regular maintenance activity and not necessarily exclusive to the realm of refactoring. We could be more equivocal in our stance if we looked at the frequency with certain refactorings had been applied to a system.

Figure 4.6 shows the trend in attribute-based refactorings applied to the HSQLDB system in the first four versions. We note that when the tool was run, version 4 was the latest available version of HSQLDB. Fifteen refactorings were extracted by the tool including: Move Field, Pull Up Field, Push Down Field and Rename Field, the description and motivation for which is as follows:

1. Move Field. "*A field is, or will be, used by another class more than the class in which it is defined. Create a new field in the target class, and change all its users*." Fowler (Fowler 1999).
2. Pull Up Field. "*Two subclasses have the same field. Move the field to the superclass.*" Fowler (Fowler 1999).
3. Push Down Field: "*A field is used only by some subclasses. Move the field to those subclasses.*" Fowler (Fowler 1999).
4. Rename Field: this refactoring is applied to make the meaning of a field clearer. It is also often undertaken after a field has been moved or pulled up/pushed down to reflect its new role Fowler (Fowler 1999).

It is noticeable from Figure 4.6 that the single 'peak' of refactorings (at version 3) occurred at the same time as the single 'peak' of net additions of attributes to the HSQLDB system shown in Figure 4.5. The highest number of refactorings was for the Rename Field and Move Field refactorings, suggesting, when also considering Figure 4.5, that classes were not necessarily 'pulled up' or 'pushed down' the inheritance hierarchy. This further implies that, in keeping with the trends described in Chapter 3, systems evolved through addition of new classes at DIT levels one and two and not necessarily through the manipulation of the existing inheritance hierarchy. We also note a coincidence of peaks of net changes in attributes and Rename Field refactorings which supports the hypothesis with respect to developer behaviour. Clearly, developers *do* refactor and, moreover, at a time when there is significant *other* regular maintenance activity taking place. Identification of when developers tend to do refactoring is currently an ongoing and open research area and this result thus gives a small insight into that behaviour.

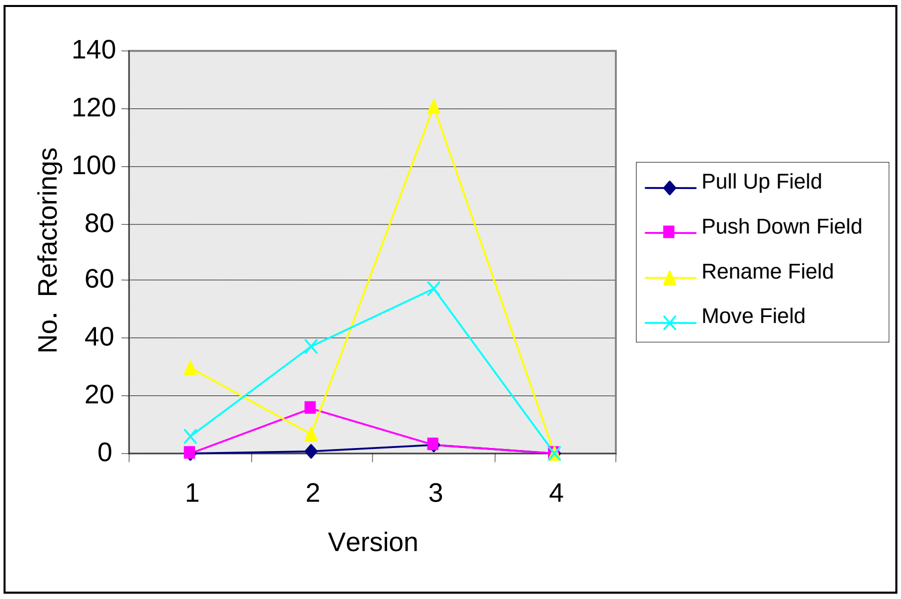


Figure 4.6. Attribute-based Refactorings for HSQLDB (4 versions)

The maximum DIT of any class in JasperReports system was 5. Figure 4.7 shows the frequencies of NOM across its twelve versions. Again, a strong tendency can be seen for the methods to be added at DIT one and to a lesser extent DIT two. Remarkably, the number of methods at DIT four and five stays constant over the course of 12 versions of JasperReports. This trend was also observed in (Nasseri et al. 2008) where the classes at DIT levels four and five showed no change in terms of number of classes in JasperReports. The overall NOM added to JasperReports was 2535, of which 2029 (i.e., 80.04%) methods were added at DIT level one and 448 (i.e., 17.67%) methods added to DIT two. The total NOM added at DIT one and two was 2477 (i.e., 97.71%). Only 58 (i.e., 2.29%) methods were added at DIT three and 0 method was added at DIT four and five.

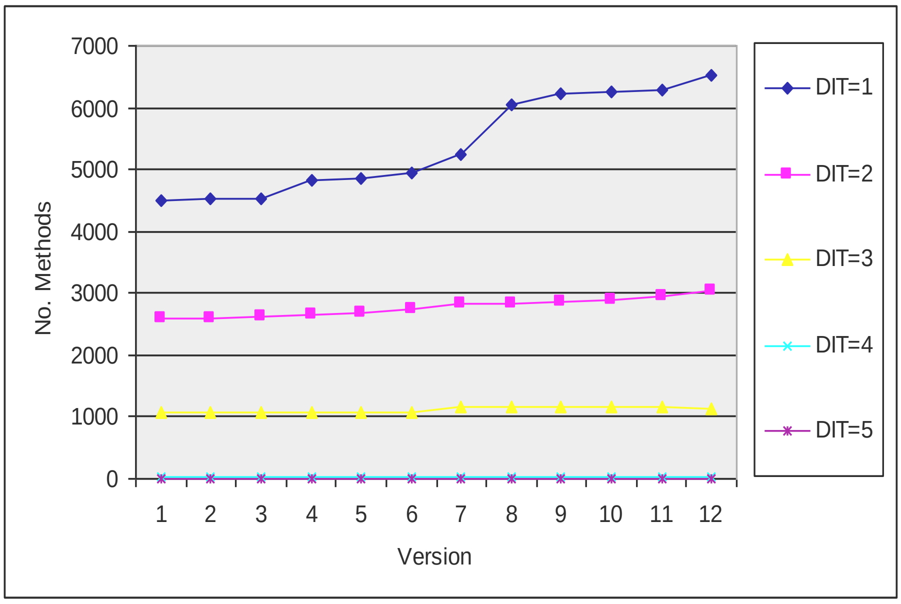


Figure 4.7. NOM frequencies JasperReports (all versions)

Figure 4.8 shows the corresponding net changes in NOM for the JasperReports system and again, shows a peak effect between both versions 3 and 4 and versions 7 and 8 (a similar trend to that of Figure 4.2 is exhibited). In the transition between version 3 and 4, 282 methods were added at DIT level one (accompanied by 36 new classes), 32 methods were added at DIT two (accompanied by 5 new classes) and only 2 methods were added at DIT three (accompanied by 2 new classes). Between versions 7 and 8, the number of classes at DIT level one rises by only 43 (i.e., 4.95%) classes. However, the NOM in the system increases by 838 (i.e., 9.08%) methods suggesting that system growth at method level was relatively higher than that at class level.

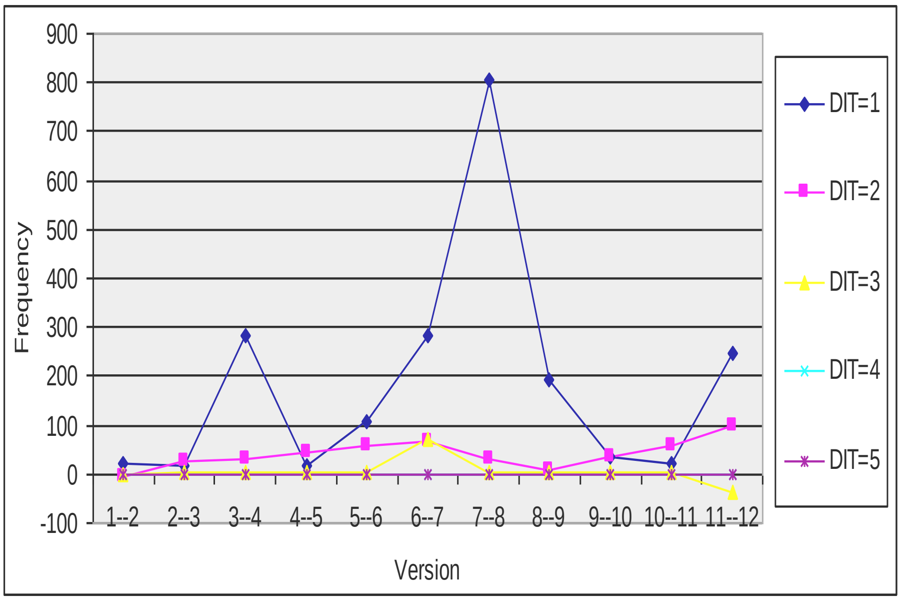


Figure 4.8. Net changes in NOM JasperReports (all versions)

Figure 4.9 shows details of the overall refactorings applied to JasperReports (when the tool was first run, version 3 was the latest available version). Once again, through inspection of the individual refactorings, we found that the refactorings were specifically related to movement of methods (and attributes). The sum of fifteen refactorings however is almost negligible compared with the large movement of methods across the versions. Nonetheless, it is a coincidence that for these first two systems, there is a direct match between a) effort invested in addition of methods and b) effort invested in refactoring. Moreover, these results are in direct contradiction with the tenet that refactoring effort should be consistently applied to a system throughout its lifetime (Fowler 1999).

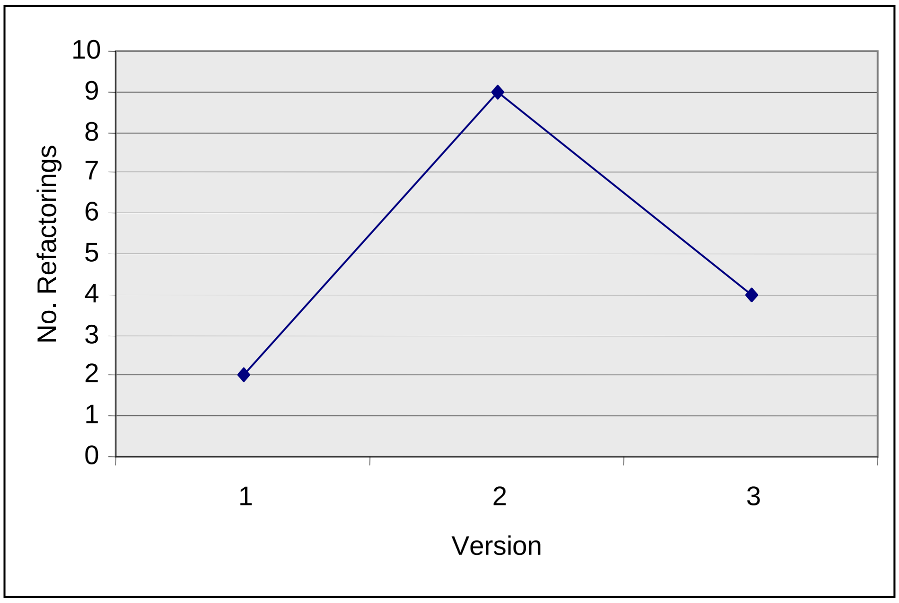


Figure 4.9. Refactorings in JasperReports (4 versions)

Figure 4.10 shows the frequencies of NOA in JasperReports. The growth rate of NOA at DIT level one is higher than that of other DIT levels. The number of attributes at DIT two also increases at initial stages of the system evolution. However, the growth rate is smaller to that of DIT one. A noticeable feature of the Figure 4.10 is that the NOA at DIT four and five stays constant throughout the entire 12 versions of the system. This trend was also observed in Figure 4.7 and 4.8 where no change was identified in NOM at DIT four and five and was also observed in (Nasseri et al. 2008) where no change was noted in number of classes at DIT four and five.

It is evident that the developers of JasperReports tended to focus their maintenance activity *only at* DIT levels one, two and three and no deeper. The total NOA added to JasperReports was 1311, of which 1274 (i.e., 97.17%) were at DIT one, and 37 (i.e., 2.83%) at DIT three. Interestingly, 177 attributes were removed from DIT two between versions 10 and 11. One suggestion for why this may have occurred is that attributes were simply moved from DIT two to DIT one as part of a conscious re-engineering effort.

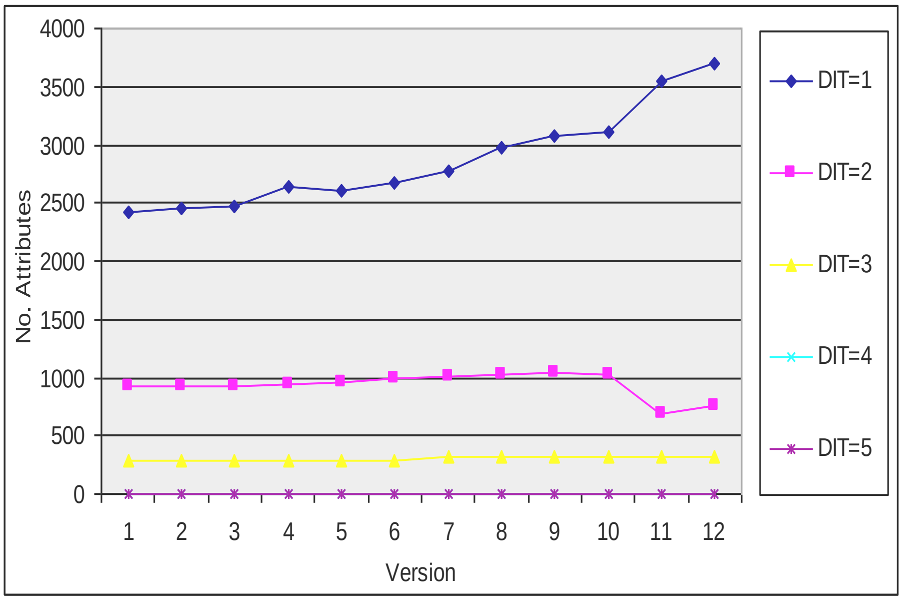


Figure 4.10. NOA frequencies JasperReports (all versions)

Figure 4.11 shows the net change in attributes through the versions of JasperReports studied. Between versions 10 and 11, there is a movement of attributes from DIT two to DIT one. From Figure 4.11, we again see a strong tendency for NOA at DIT level one to fluctuate.

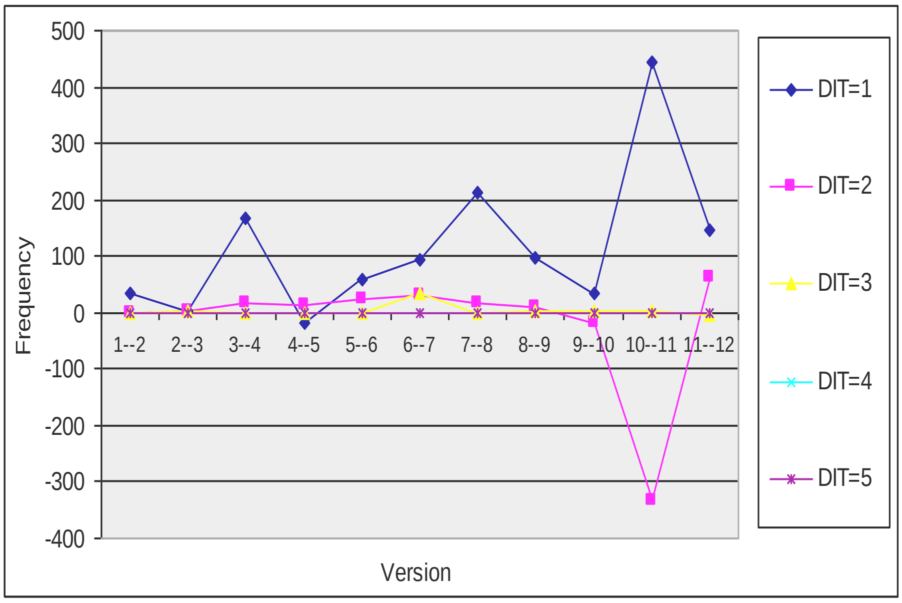


Figure 4.11. Net changes in NOA JasperReports (all versions)

Figure 4.12 shows the net changes of classes at all DIT levels in all versions of JasperReports and shows changes in every version. The maximum change (58 classes) occurs between versions 6 and 7. In terms of net changes in NOA, the maximum change of NOA (231 attributes) occurs between versions 7 and 8 suggesting that evolution at different levels of granularity may again, as per HSQLDB, show a different trend (cf. Figure 4.11 and 4.12).

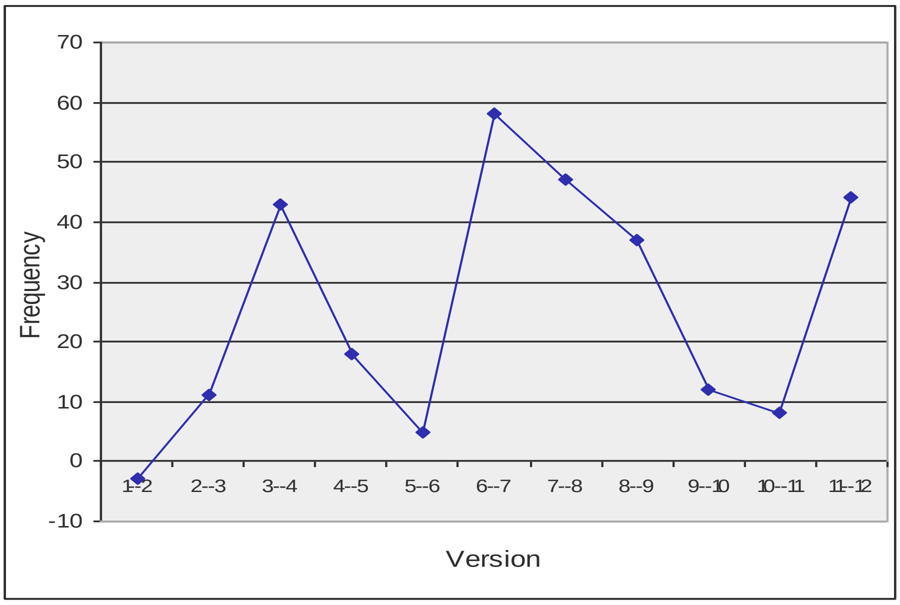


Figure 4.12. Net changes in classes JasperReports (all versions)

Table 4.4 shows the number of net changes of classes at different levels of DIT in JasperReports. In keeping with the HSQLDB system, there appears to be a lack of addition of classes in earlier versions of the system. One plausible theory for this feature is that there was a time 'lag' between when the system was first released and the signs of decay. That decay is accompanied by a concerted re-engineering effort.

Version	DIT 1	DIT 2	DIT3	DIT4	DIT5	Total
1-2	-2	-1	0	0	0	-3
2-3	3	6	2	0	0	11
3-4	36	5	2	0	0	43
4-5	9	9	0	0	0	18
5-6	11	-6	0	0	0	5
6-7	35	18	5	0	0	58
7-8	43	4	0	0	0	47
8-9	33	3	1	0	0	37
9-10	5	6	1	0	-1	11
10-11	-1	9	0	0	0	8
11-12	33	9	2	0	0	44

Table 4.4. Net class additions JasperReports (All versions)

The same four refactorings (attribute related) for the first three versions of JasperReports are shown in Figure 4.13. We note that when the refactoring tool was run, version 3 was the latest available version for JasperReports. Only two of the four refactorings are non-zero. No evidence of either of the ‘Pull Up Field’ or ‘Push Down Field’ refactorings were found in any of the versions of this system. The fact that there was also peak of added attributes coinciding with that in Figure 4.11 supports the hypothesis that significant effort was applied to the system at this point and that refactoring effort did coincide with that effort. Again, this gives us an insight into the question as to whether developers *do* refactor and more importantly ‘when’ they refactor.

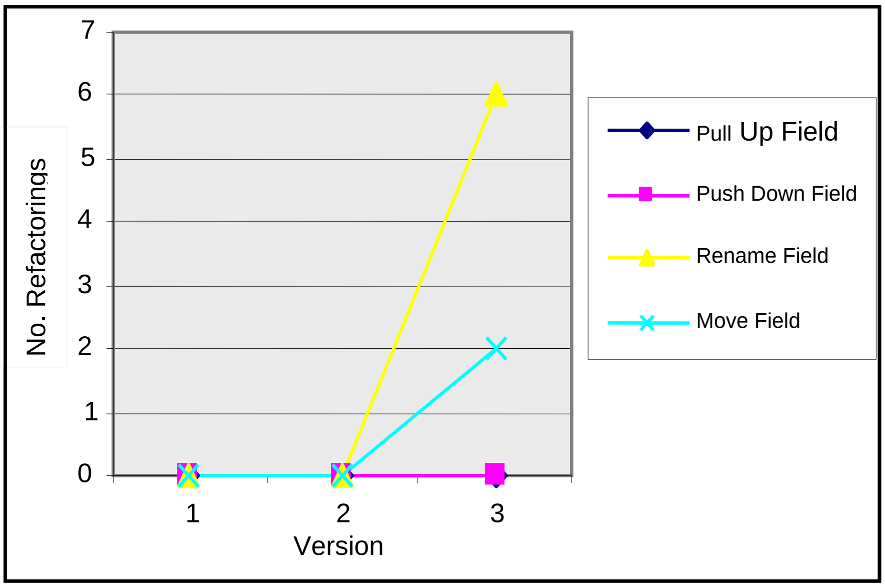


Figure 4.13. Attribute-based Refactorings for JasperReports (3 versions)

The maximum DIT of any class in Tyrant system was 5. Figure 4.14 shows the frequencies of NOM at DIT levels one, two, three and four in the Tyrant. In total, 92.31% of all methods were added at DIT levels one and two, and only 7.69% were added at DIT levels three, four and five (level five is not shown in the Figure). Remarkable is the dramatic rise from version 4 to version 5 at DIT level one. As noteworthy is the fall in the NOM between these versions at levels two and three. Even more pronounced is the fall in the number of classes at DIT level four where from 281 methods in version 4, zero methods were found in version 5 (at DIT level five, the 16 methods in version 4 fell to zero methods in version 5). For this system, there appears to have been an extensive re-engineering effort after version 4. The pattern seems to have been to remove classes from lower levels (levels four and five) of the hierarchy and place them at higher levels. This pattern would fall into line with the 'Extract Method' and 'Extract Superclass' refactorings (Fowler 1999) or even one of Fowler's 'Big Four' refactorings – the 'Collapse Hierarchy' refactoring (Fowler 1999).

As noticeable from Figure 4.14 is the increasing trend from versions 26 to 37 at DIT level one, which suggests that after a period of relative stability, more re-engineering was applied to this system. The question that naturally arises is whether this movement of methods was accompanied by a corresponding movement of classes? In other words, was the addition of methods from existing functionality or from added functionality?

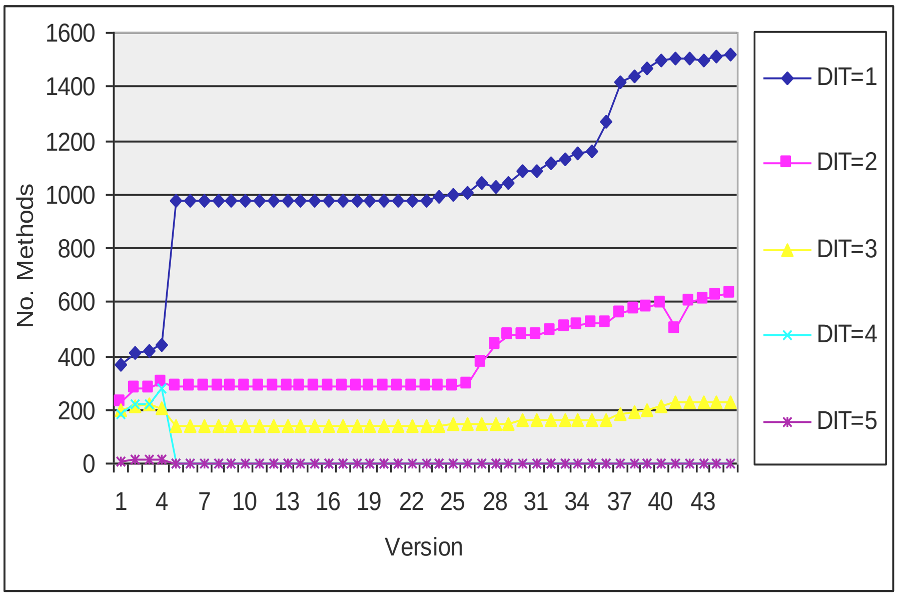


Figure 4.14. NOM frequencies Tyrant (all versions)

Figure 4.15 shows the net changes in classes at DIT level one for the Tyrant system. We can see that the trend in number of methods contrasts with the net change in the number of classes. The number of classes in the Tyrant system remains relatively static, and there is only a relatively small rise in number of classes from 143 to 172 (29 classes) between versions 4 and 5. Most noticeable from Figure 4.15 are the peaks between version 4 to 5 and version 35 to 37 corresponding to the rise in methods shown in Figure 4.14. This was a surprising result from the analysis and suggests that classes were not only being moved up the hierarchy, but that classes may have had all their methods moved from one class and merged with other classes. This operation is the opposite of the 'Extract Class' refactoring and would be detrimental to the overall cohesion of the system (Briand et al. 1998); merging classes tends to dilute their original purpose.

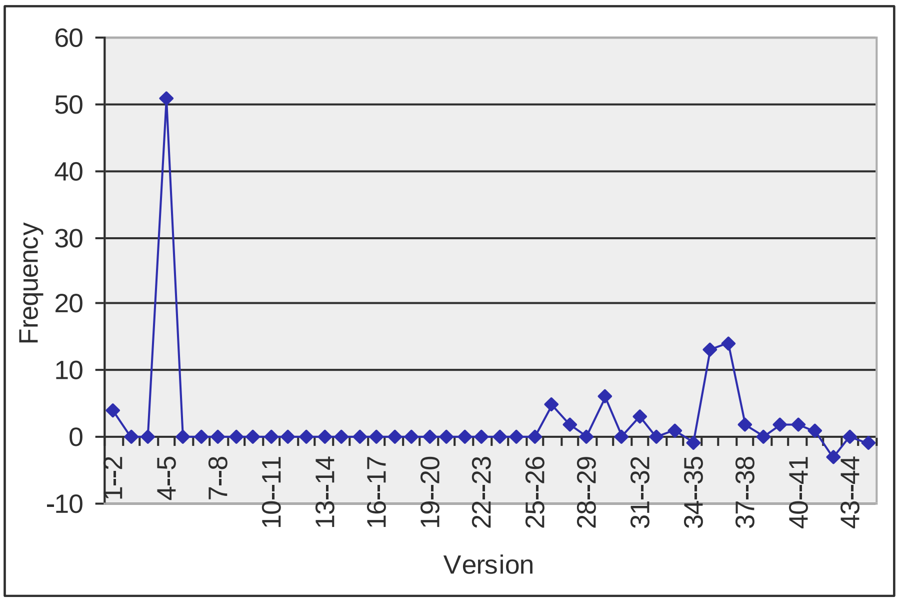


Figure 4.15. Net changes in classes DIT 1 Tyrant (all versions)

Figure 4.16 shows the refactorings applied to versions 1 to 9 for Tyrant extracted by the same tool used for the HSQLDB and JasperReports system (N.b., when the tool was run version 9 was the latest available version of the Tyrant). It is noticeable in the case of Tyrant that there is a single peak of refactorings which occurred close to the peak in additions of methods to the system shown in Figure 4.14. Once again, when we inspected the actual refactorings identified by the tool, we found evidence of refactorings related to movement of methods and attributes. The most revealing feature of Figure 4.16 when compared with Figure 4.14 is that the refactorings seemed to have been undertaken *after* the sudden rise in NOM. This suggests that, unlike for the HSQLDB and JasperReports systems, where there was a coincidence between refactorings and the addition of methods, the motivation for refactoring in this system was different. It appears that the effort shown in Figure 4.16 may have been an ‘effect’ corresponding to the ‘cause’ shown at version 4 in Figure 4.14. Put another way, the refactoring process may have acted as a ‘tidying up’ mechanism after the sudden burst of maintenance activity.

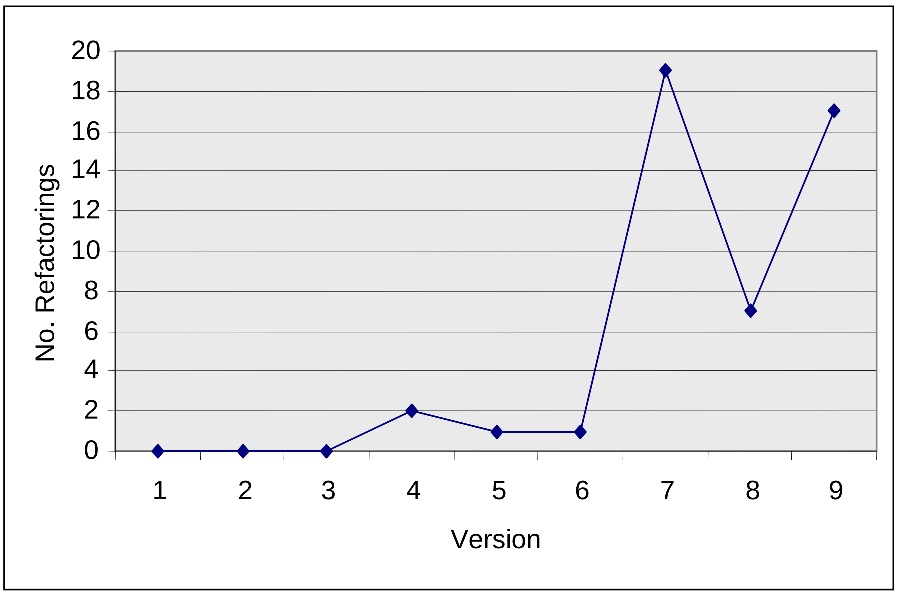


Figure 4.16. Refactorings in Tyrant (9 versions)

Figure 4.17 shows the frequencies of NOA at DIT levels one, two, three and four in the Tyrant system. In (Nasseri et al. 2008) we observed that the maximum DIT in Tyrant dropped from 5 to 3 in version 5. The number of classes at DIT level one increased from 45 to 96 in version 5 and the system exhibited no change in terms of number of classes until version 26. Similarly, in Nasseri and Counsell (Nasseri and Counsell 2009b), it was observed that the NOM at DIT level one increased from 437 in version 4 to 978 in version 5 where, again, the system exhibited no change in terms of number of methods until version 23. From Figure 4.17, the NOA at DIT level one increases from 535 in version 4 to 590 in version 5 which is not significant compared to the increase in number of classes and methods. Had we used only classes and methods for our analysis, we would have overlooked an important trend in the evolution of this system. In Tyrant, 742 attributes were added overall, of which 512 (i.e., 69%) attributes were added at DIT level one (the number of attributes at DIT two and three decreased by 57 and 67, respectively).

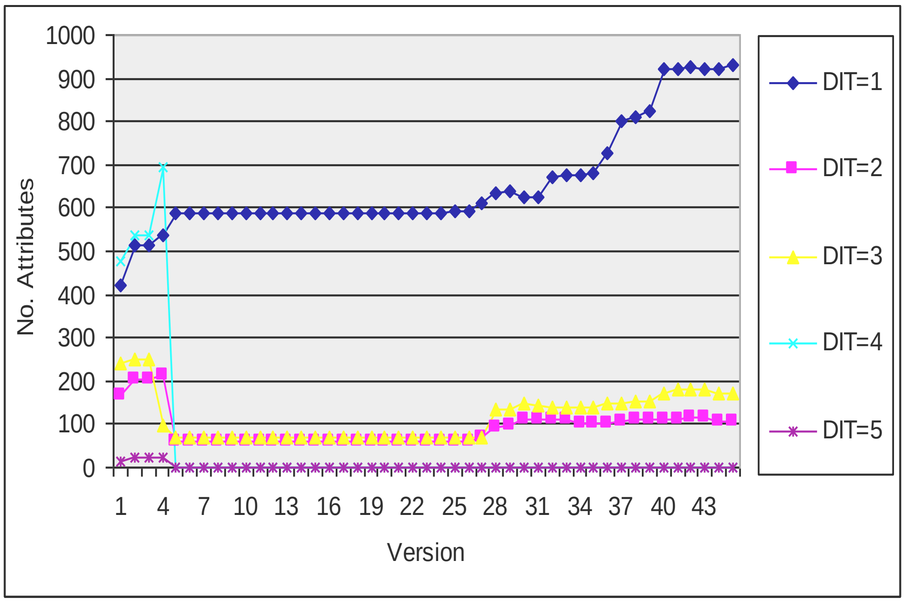


Figure 4.17. NOA frequencies Tyrant (all versions)

From version 1 to version 4, 219 (i.e., 29.51%) attributes were added at DIT four and 11 (i.e., 1.48%) added at DIT level five. However, after version 5 the classes from DIT four and 5 were removed.

Figure 4.18 shows the net changes of attributes in Tyrant. Since the number of classes at DIT four and five falls to zero in version 5, we excluded attributes at DIT four and five from the figure. We see that the net changes in number of attributes at DIT level one are predominantly positive. In version 4 of Tyrant, the NOA at DIT two and three falls with a corresponding increase in number of attributes at DIT one. The most notable feature for this system is the fact that after version 4, where maximum DIT drops from 5 to 3, the system stabilizes and thereafter no change in number of classes, methods or attributes is made to the system for the duration of a number of versions. In Tyrant, the total number of removed attributes at DIT levels one, two and three are -14, -169 and -197, respectively throughout the entire 45 versions of the system. This implies that the number of added attributes at DIT level one were significantly higher than the number of removed attributes at this level. In contrast, the number of added attributes at DIT levels two and three tended to be significantly lower than the number of removed attributes in these levels. This latter result again implies that while new attributes are added at DIT level one, some attributes may have been moved from DIT levels two and three to level one as a result of refactoring (possibly using Pull Up Field and Pull Up Method).

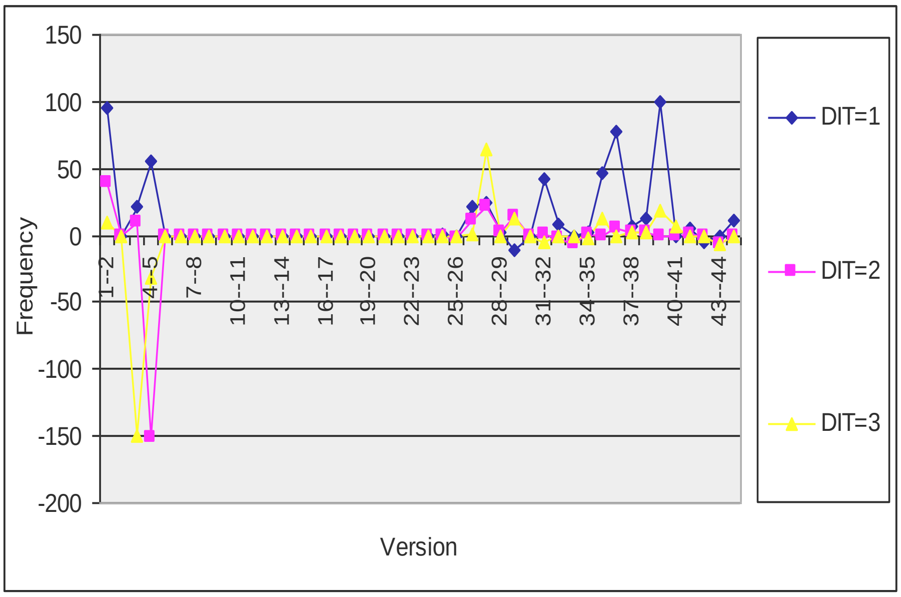


Figure 4.18. Net changes in NOA Tyrant (all versions)

Figure 4.19 shows the net changes of classes in Tyrant. The system stabilizes after version 4 where significant change is made to the system. We believe the system underwent re-engineering activity and, as a result, system 'stability' was improved. Furthermore, the maximum change in number of classes in Tyrant (+29) occurred between versions 4 and 5. In terms of changes of NOA, the maximum change (-848) occurred between the same versions (4 and 5). Once again, evolution at lower granularity (i.e., attributes) can show a different and opposite trend in systems' evolution.

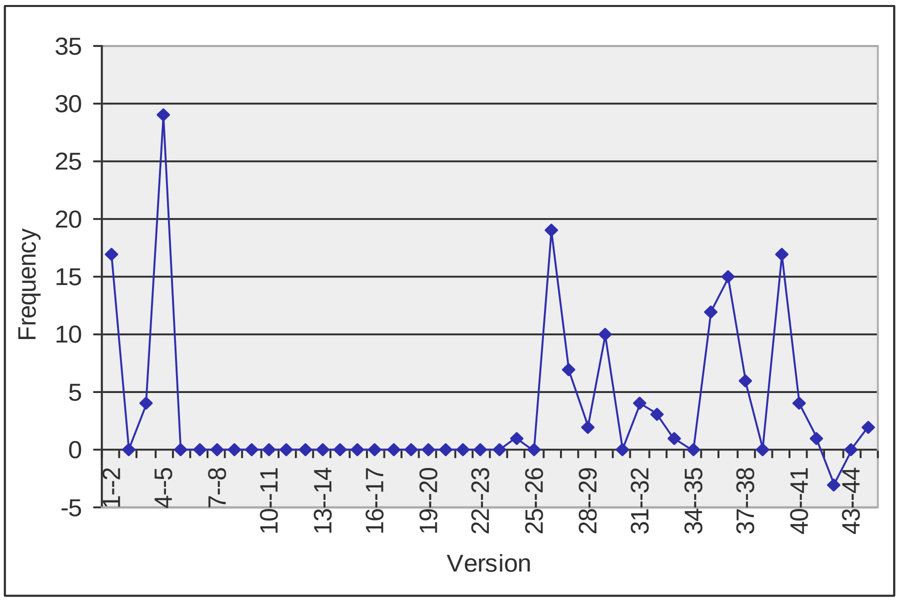


Figure 4.19. Net changes in classes Tyrant (all versions)

Figure 4.20 shows the same four refactorings for Tyrant (as was presented for HSQLDB and JasperReports). In keeping with the other two systems, very few refactorings were undertaken for this system across the versions studied. (When the refactoring tool was run, version 9 was the latest available version of Tyrant.) We do see some evidence of renaming of attributes at later versions of the system, but this would seem to be more related to moving of existing class features than addition of new.

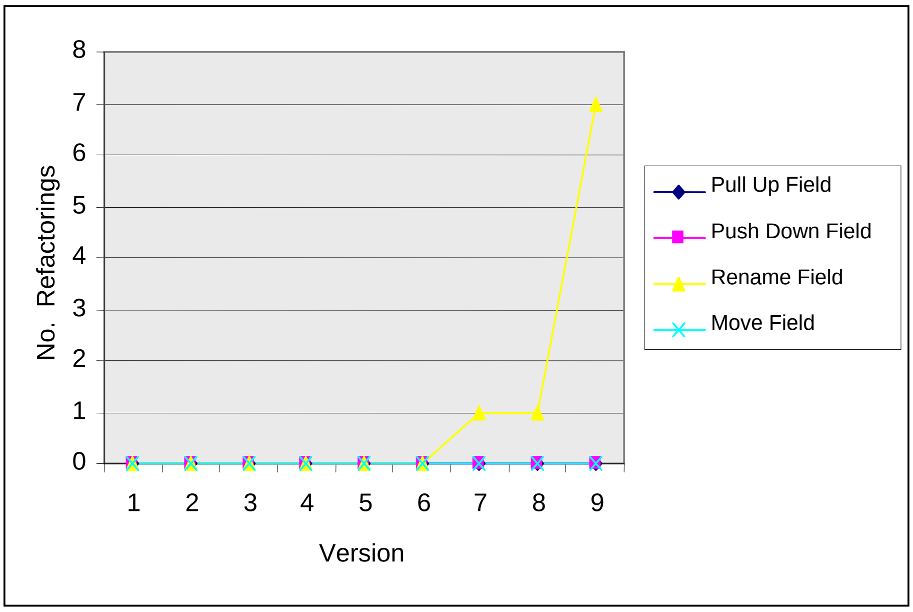


Figure 4.20. Attribute-based Refactorings for Tyrant (9 versions)

Figure 4.21 shows the total changes of attributes (both positive and negative) at each DIT level in HSQLDB, JasperReports and Tyrant. It is evident that similar to changes of classes (Nasseri et al. 2008) and methods (Nasseri and Counsell 2009b), the vast majority of changes of attributes also occurred at DIT level one. While we concede that the majority of new attributes were made where the majority of functionality could be found, there are many sub-plots to that trend which a project manager and developer would find useful.

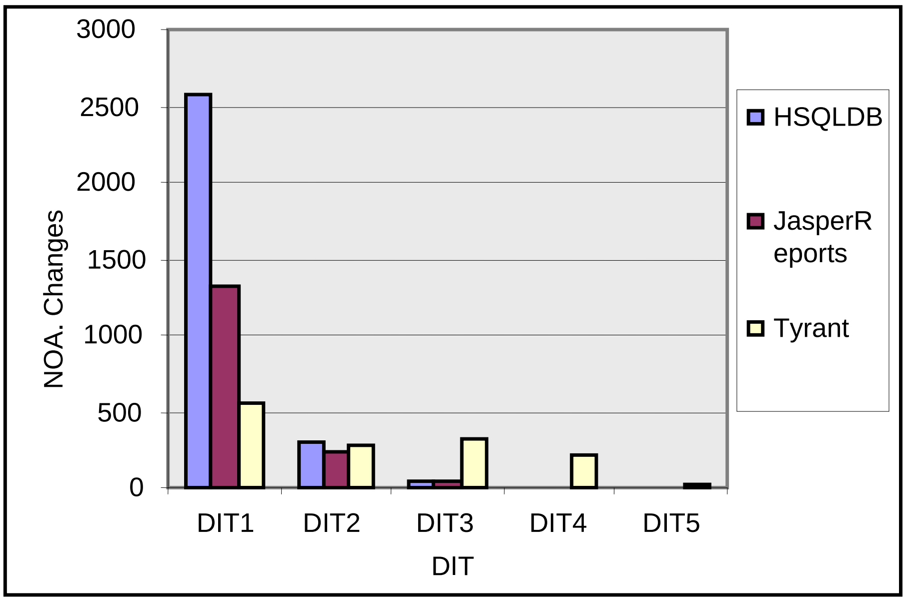


Figure 4.21. Total change in NOA (3 systems)

4.4.1 Deeper levels of inheritance

One question that arises from the preceding analysis is whether in systems with deeper levels of inheritance, the same mismatch between trends in methods, attributes and classes arises. To answer this question, we collected the same method and attribute data from SwingWT. The maximum DIT of any class in SwingWT system was 7. Figure 4.22 shows the frequencies of the NOM at each DIT level for SwingWT. From Figure 4.22, the NOM at DIT level one has the highest growth rate. The total NOM added in SwingWT was 6578, of which 3506 (i.e., 53.30%) methods were added at DIT level one and 853 (i.e., 12.97%) added at DIT level two.

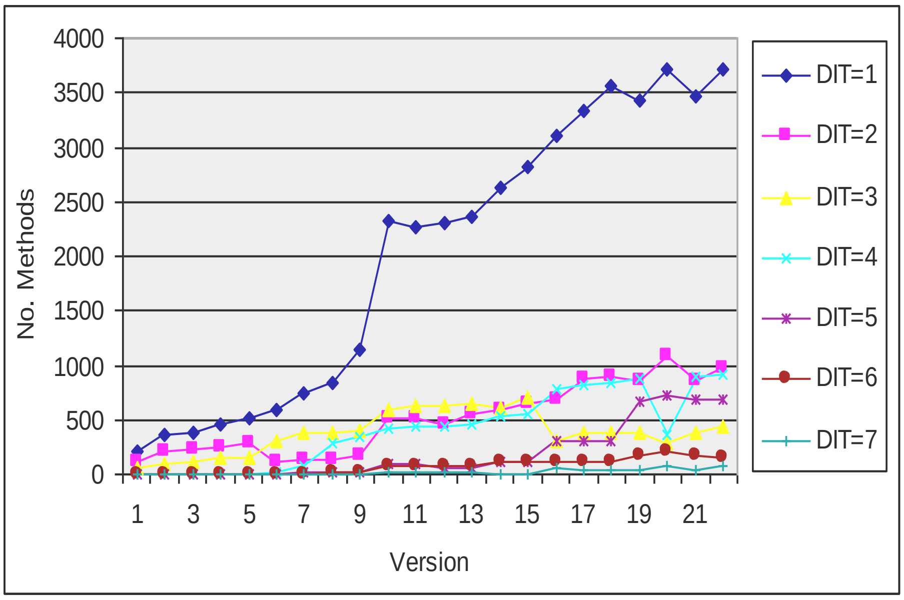


Figure 4.22. NOM Frequencies SwingWT (all versions)

Figure 4.23 shows the net change in methods for the SwingWT at each DIT level. From Figure 4.23, a peak in changes of methods around version 9 is visible. Similarly, between version 15 and 16, a negative peak can be seen in number of methods at DIT three. Figure 4.24 shows the net changes of classes in SwingWT. From Figure 4.24, every change in number of classes is positive.

Figure 4.23 shows a different trend to that of Figure 4.24. There are some negative changes as well as positive changes in NOM in SwingWT. This was an interesting feature to emerge from our analysis of SwingWT, supporting our view that analysis at the method and attribute level can often indicate different perspectives to a similar analysis at the class level. The over-riding implication is that since the remaining methods were likely to be distributed around classes at levels one, two and three, then we might expect such large-scale changes to be a potential source of faults in later versions.

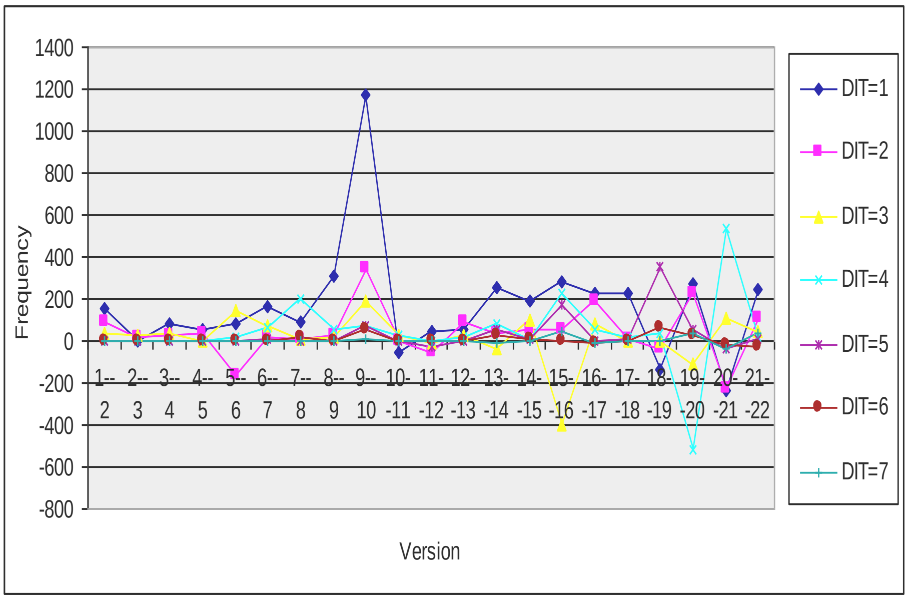


Figure 4.23. Net changes in NOM SwingWT (all versions)

In other words, a widely distributed and scattered set of changes is likely to be more problematic than making changes in a targeted and localised area, simply through the potential for side-effects at *each* of those disparate locations where changes were made.

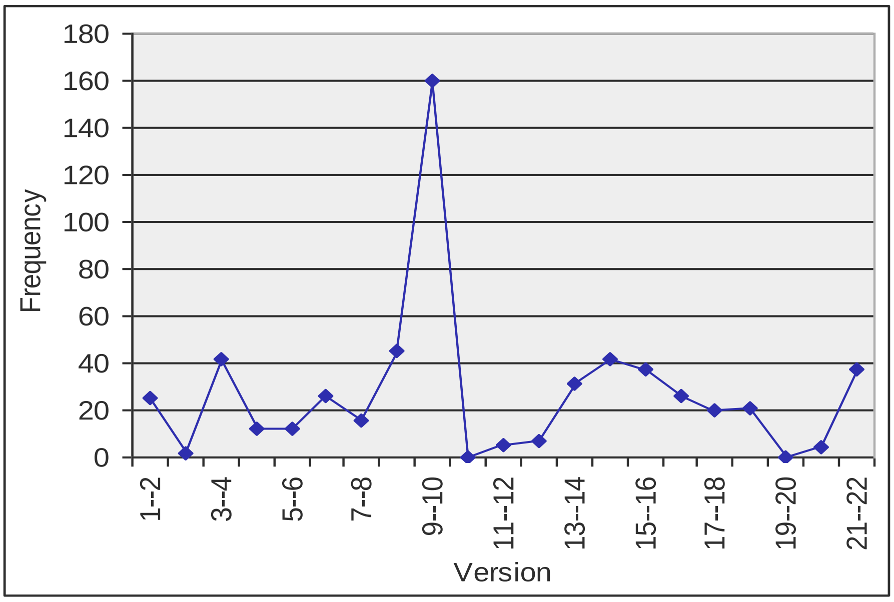


Figure 4.24. Net changes in classes SwingWT (all versions)

The likelihood of SwingWT exhibiting faults as a result of these identified patterns is made more likely by the fact that Swing system has been the subject of many empirical studies (Advani et al. 2006, Counsell et al. 2006a); in each case, practices used in the system have been completely at odds with sound and sensible OO practice. Overall, the mismatch between net added classes and net added methods would seem to reflect a poor and patchy evolution strategy for this system at best. At worst, evolution of this system seems to imply a consistent storing up of problems for subsequent versions.

A key motivation for analysing the evolution of systems at the method level was to try to identify evolutionary forces at low granularity that could not be extrapolated at the class level and the analysis of trends in the SwingWT system is a case in point. Figure 4.25 shows the frequencies of NOA at each DIT level for SwingWT. From Figure 4.25, attributes at DIT level one tend to have the highest growth rate in this system. The number of methods (Figure 4.22**)** and classes in SwingWT also showed a similar trend in (Nasseri et al. 2008, Nasseri and Counsell 2009b). The total number of attributes added to SwingWT was 2282, of which 1306 (i.e., 57.23%) attributes were added at DIT level one and 260 (i.e., 11.40%) attributes added at DIT level two.

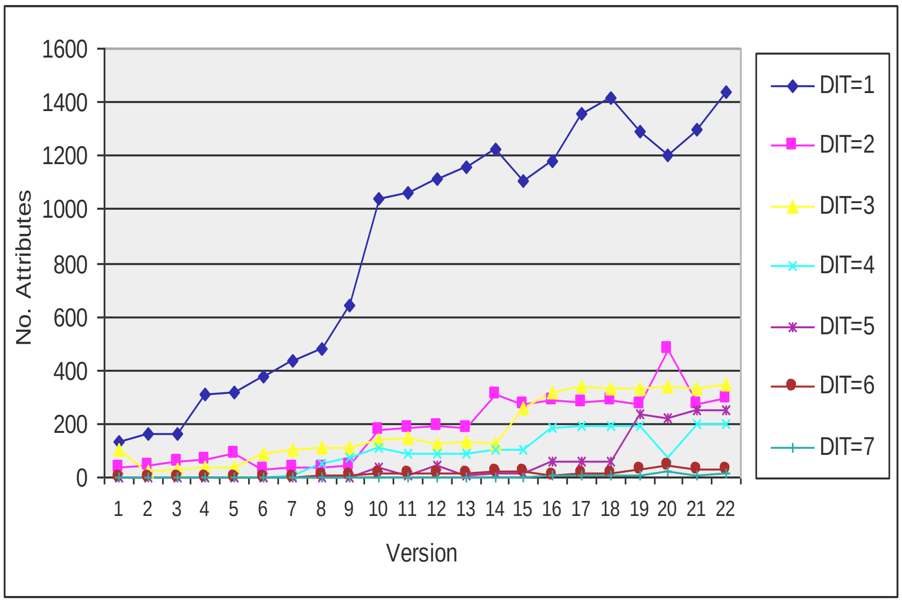


Figure 4.25. NOA Frequencies SwingWT (all versions)

Figure 4.26 shows the net changes of attributes in SwingWT. We see a peak in version 9 of the system. This trend was also observed in changes of methods in (Figure 4.23) and in changes of classes in (Figure 4.24). In version 9 of SwingWT, the number of attributes increases by 645, accompanied by 1929 methods and 160 classes.

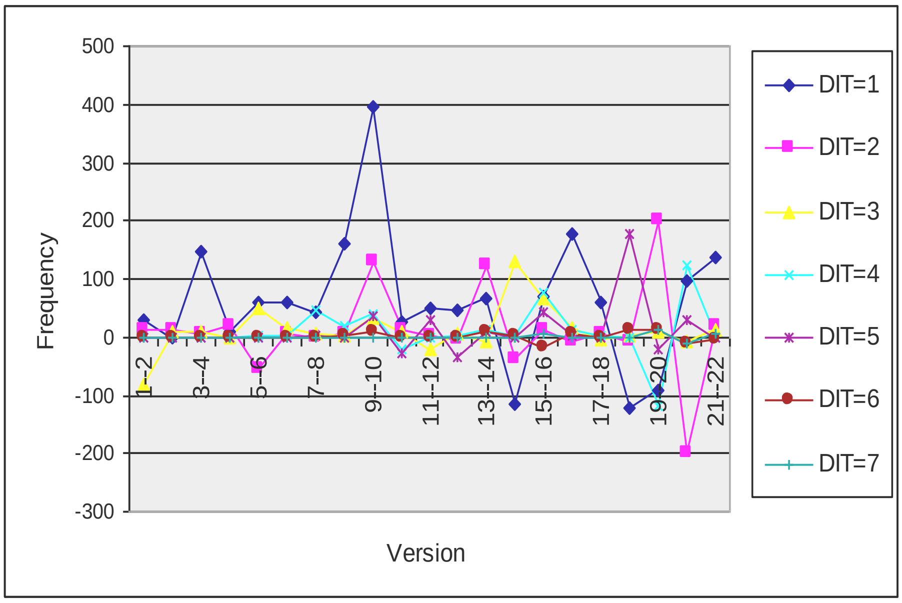


Figure 4.26. Net changes in NOA SwingWT (all versions)

From Figure 4.24, it is interesting that the changes in number of classes in SwingWT is always positive, suggesting the growth of the system in every consecutive version. The transition between version 9 and 10 seems to be the point where the maximum classes (160) were added to the system. This trend can also be seen in Figure 4.26 where the maximum change in NOA (645) occurs in the same transition of versions.

4.5 Discussion

A number of issues are raised by the analysis in this chapter. Firstly, we have looked at different numbers of versions of each of the four systems. While ideally, we would have liked to have had the same number of versions for each system, we wanted to extract as much information about available data as possible. The number of versions of the systems studied is different in each system (i.e., minimum number of versions studied was 6 and the maximum 45). We feel that the sample is small but does provide an insight into features that might be found in many other OSS. Secondly, we have assumed an equal time ‘gap’ between the versions of each system. In practice there are likely to be widely varying time gaps between versions and different motivation for the activities between versions (e.g., fixing faults, enhancements). While this could influence the results presented, the overall theme running through our analysis would remain. From that point of view, we feel we can defend such a criticism. In a previous study (Nasseri et al. 2008) and in this chapter we observed that developers of OSS tend to invest the majority of their maintenance activity at DIT level one and, to a lesser extent, level two. There are a number of possible explanations for this. Most of the changes may be pertinent to the classes located at higher levels of DIT simply because that is where most of the functionality is located (i.e., it is a self-perpetuating phenomenon). Developers may want to avoid the burden of understanding classes located at deeper levels of the inheritance hierarchy when making changes. This may not be their fault. Since developers of OSS are geographically separated, the design documentation for OSS is not always accessible or available by every developer at every stage of maintenance.

4.6 Summary

A common feature of many OO evolution studies is to use classes as a basis. While classes provide a tried-and-trusted basis, there may be many features of an evolving system that occur at the lower-level of methods and attributes. In this chapter, we investigated trends in four Java OSS at this level to determine whether this was the case. A number of insightful features of the four systems were revealed, which in many cases, could not have been extrapolated through a class analysis alone. For example, we observed that in some cases evolution of a system at the method and attribute levels show a different trend to a similar analysis at the class level. Moreover, analysis of a system at the method and attribute level allows us capture the level of granularity which may be missed when analyzing the systems at the class level. A comparison between refactoring data from a prior study and the method and attribute level evolution allowed some relevant comparisons to be made. The comparisons enabled us to observe the peaks in refactoring activity compared with changes in number of methods and attributes in the four systems. The research highlights the benefit of firstly analysing the evolution of systems at a finer-grain and secondly, of using previous studies to inform current results. The benefit to developers of the study is that it will allow an understanding of how a Java OSS evolves at the method and attribute level in comparison with class evolution, and where remedial effort at different levels of granularity may need to be applied in future versions.

CHAPTER 5 Class Movement and Re-location

5.1 Introduction

In the previous chapter, we investigated the trends in the evolution of four Java OSS at the method and attribute level, and compared those low-level changes to a set of low-level refactorings (refactorings pertaining to methods and attributes) applied to initial versions of the systems. In this chapter, we manually inspect the position of a large sample of classes in the inheritance hierarchy of multiple versions of four Java OSS. The DIT inheritance metric was extracted from the systems using the JHawk tool. We examined the position of each class in the inheritance hierarchy as each of the systems evolved. This allowed us to construct a pattern of the classes that tended to be moved, where they moved to, and when (in terms of version number). The work described in this chapter thus extends previous work by the same authors, (Nasseri et al. 2008) and described in Chapter 1, to a finer-grain; it does not simply look at overall patterns of increases or decreases in classes as a system evolves, but actually *where* the activity takes place. Thus the research question for the study in this chapter is: in the context of the tendency of systems to deteriorate in quality as they evolve *and* to exhibit shallow levels of inheritance depth, what observable, evolutionary patterns can be determined from class addition, deletion and movement around the inheritance hierarchies of systems?

In the next section, we present the motivation for the research in this chapter. Section 5.3 provides study design including the four OSS used and data collected. We then present data analysis including class movement and re-location analysis and an analysis of class characteristics (Section 5.4). In Section 5.5, we provide a discussion of the study before we present a summary of the empirical study (Section 5.6).

We also acknowledge that a part of this chapter has been accepted for publication (subject to minor changes), in (Nasseri et al. 2009)

5.2 Study Motivation

Our research in this chapter is to highlight trends and features of an inheritance hierarchy as it evolves from the perspective of class re-location. Such a study can inform decision-making by a developer or project manager if those trends show any clear patterns. For example, observation of a certain subset of classes being consistently moved together (or duplicated) around the hierarchy suggests that the subset might need to be amalgamated for ease of reuse or simply refactored using the Collapse Hierarchy refactoring, for example (where a subset of classes are merged) (Fowler, 1999). Equally, if a single class containing relatively high amounts of coupling is being moved around frequently, then again, it might need to be decomposed or a permanent home found for it. In addition, the analysis of class evolution makes it possible to identify the most change-prone parts in the systems and for remedial, re-engineering action to be taken as a result.

5.3 Study design

5.3.1 The four open-source systems

For the study in this chapter we used four Java OSS from our system archive to investigate the class movement and re-location. The systems used were: HSQLDB, JasperReports, SwingWT and Tyrant (see systems 1, 3, 5 and 8 described in Section 2.3.2.5). Since the number of versions in some systems was large and the analysis onerous, we chose to analyse the first, followed by every fifth and final version of each system. For this study we thus used versions 1, 5 and 6 of HSQLDB, versions 1, 5, 10 and 12 of JasperReports, versions 1, 5, 10, 15, 20 and 22 of SwingWT and versions 1, 4, 5, 10, 15, 20, 25, 30, 35, 40 and 45 of Tyrant. The extent of changes in the Tyrant system at version 4 was a feature not shared by any of the other three systems and hence justifies our choice of its inclusion.

5.3.2 Data collected

The JHawk tool was used to automatically collect the following measures from the versions of the four systems: Depth of Inheritance Tree (DIT), Number of Methods (NOM), Lack of Cohesion Of the Methods in a class (LCOM) and Message Passing Coupling (MPC) (see metrics 1, 5, 9 and 10 in Section 2.3.2.6).

The DIT metric was used to determine patterns in new, deleted and moved classes. The remaining three metrics were used to determine the characteristics of those classes. Having collected the metrics, we manually analyzed the position of each class in the inheritance hierarchies for each system according to the versions specified in Section 5.3.1. We were then able to categorise changes in the form of a) new classes b) deleted classes and c) moved classes in each of the inheritance hierarchies. For each class falling into one of these three categories, we then investigated whether class characteristics i.e., class size and coupling exhibited any particular or remarkable characteristics.

5.4 Data analysis

5.4.1 Class Movement and Re-location Analysis

We start our analysis of class movement and re-location with the HSQLDB system. This system started with only 56 classes of which 54 classes were at DIT one and only 2 classes at DIT level two. The maximum DIT in the entire 6 versions of HSQLDB reached 4. Figure 5.1 shows the frequencies of number of classes at each DIT level in versions 1, 5 and 6. From Figure 5.1, we see that DIT one is where the majority of classes were added. The number of classes at DIT one reached 279 in version 6. The number of classes at DIT two and three also increased but to a far lesser extent.

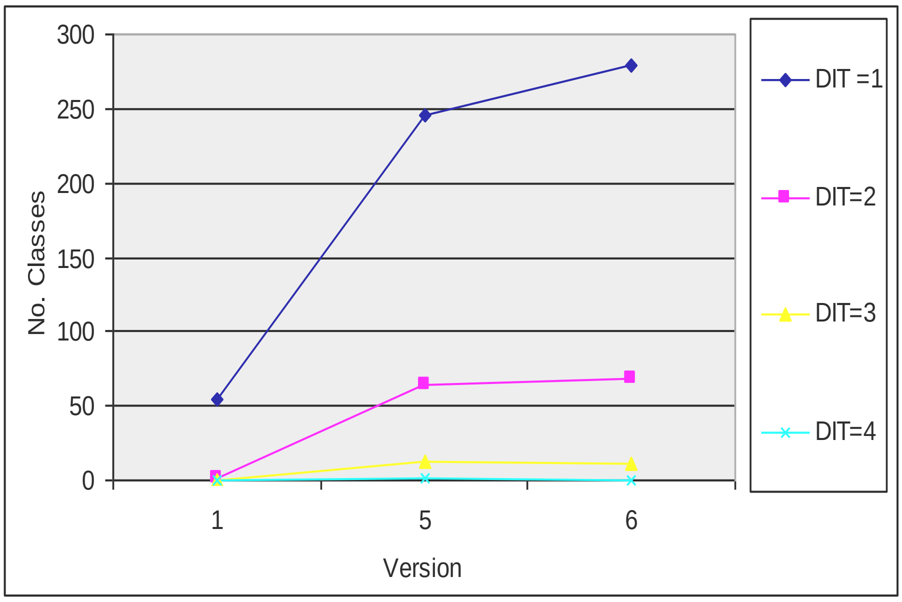


Figure 5.1. DIT frequencies in HSQLDB (3 versions)

The values from Figure 5.1 show only the net increases or decreases in number of classes. On that basis, the following questions arise. Are all of the added class new? Have any classes been deleted? Most importantly for the thrust of the research in this chapter, is there any movement of classes across inheritance hierarchy (or do classes tend to stay largely where they are)?

Table 5.1 shows the data for the movement of classes within the inheritance hierarchy between versions 1 and 5 in HSQLDB. Table 5.1 also shows the number of removed (RemC.) and the number of new classes (NewC.) added at each DIT level in the same transition. For Example, 3 classes were moved from DIT one to DIT two, 8 classes were removed from DIT one and 203 new classes added to this level. Evidence confirms the view that while most activity in terms of new classes seems to happen at DIT one, there are certain occurrences of classes being pushed down the hierarchy (although usually in small numbers). We can only suggest that the developers moved classes from DIT one to DIT two so that those ‘moved’ classes could take advantage of functionality offered through inheritance by classes at DIT one.

	DIT1	**DIT2**	**DIT3**	**DIT4**	**RemC.**	**NewC.**
DIT1	•	3	0	0	8	203
DIT2	0	•	0	0	0	59
DIT3	0	0	•	0	0	12
DIT4	0	0	0	•	0	1

Table 5.1. Class evolution between version 1 to 5 in HSQLDB

From visual inspection, a new class was added above one of the moved classes at DIT one; the other two classes were added as subclasses of two existing classes at DIT one. The total number of new classes in the system was 267 and only 8 classes were removed from DIT level one; those eight classes were not re-located and were deleted from the system. It is possible, however, that the 203 added classes might have included those same 8 classes re-located but with a new name, simply amalgamated to form new classes or integrated into other classes. If any of these cases applied, then one suggestion is that these classes may have been the target of refactoring effort (Fowler, 1999), through use of renaming, decomposition of classes or collapsing of sub-hierarchies (evidence of which we found in other systems). Table 5.2 shows the data for the movement of classes across the inheritance hierarchy between versions 5 and 6 in HSQLDB, in the same format as Table 5.1.

	DIT1	**DIT2**	**DIT3**	**DIT4**	**RemC.**	**NewC.**
DIT1	◆	1	0	0	8	39
DIT2	3	◆	0	0	0	5
DIT3	0	1	◆	0	0	0
DIT4	0	0	0	◆	1	0

Table 5.2. Class evolution between version 5 to 6 in HSQLDB

In version 6, the total number of classes in the system increased from 323 to 358. One class was moved from DIT one to DIT two and 3 classes were moved from DIT two to DIT one. One new class was added above the moved class from DIT one to DIT two, suggesting that the addition of this new class could be part of an ‘Extract Superclass’ refactoring (Fowler, 1999). The 3 displaced classes from DIT two were separated from their respective superclasses and moved to DIT one. There seems to be evidence of movement of classes from DIT two to DIT one from the data illustrated so far.

The maximum DIT in version 5 was 4 and only one class could be found at this level. In version 6, the same class was removed thereby reducing the maximum DIT in this version to 3. From Table 5.2, we also note that there is very little movement of classes within the inheritance hierarchy. The majority of changes are incremental (i.e., new classes) suggesting that for this system, a well structured inheritance hierarchy was in place and that lent itself well to the addition of classes. Figure 5.2 shows the frequencies of number of new classes (NewC), removed classes (RemC) and moved classes (MovC) in the studied versions of HSQLDB.

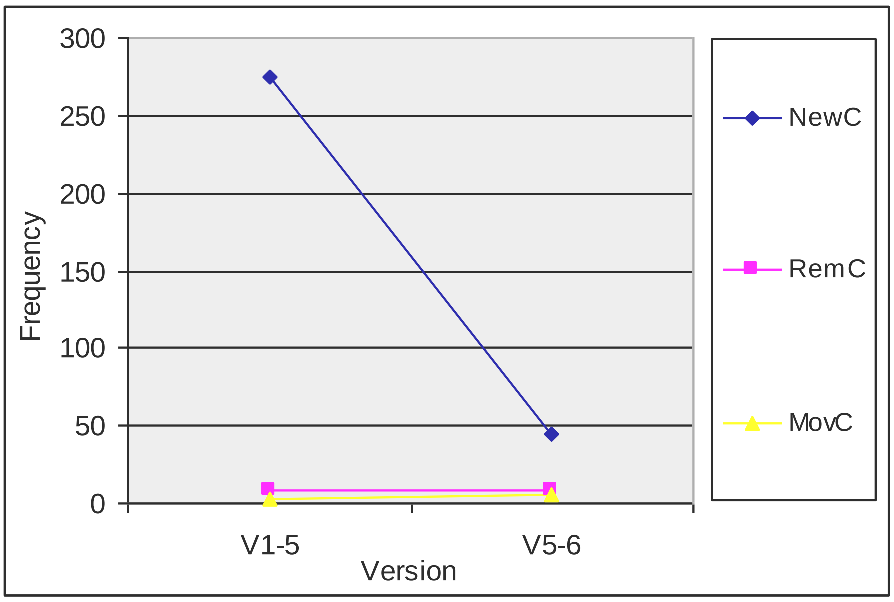


Figure 5.2. Changes in HSQLDB

Since the full set of documentation for an OSS system is not always available, we believe that OSS may often be maintained based on a model such as the *Iterative-Enhancement* (I-E) maintenance model of Basili (Basili 1990). That model is usually used for maintenance of proprietary systems when the full set of requirements is not fully understood by developers. The underlying principle of the I-E model is to re-design, reuse and/or replace parts of an existing system that is exhibiting features rendering it difficult to maintain. Evidence presented so far suggests this might be an appropriate model for systems which evolve in a haphazard fashion.

The JasperReports system started with 818 classes in version 1 and contained 1098 classes by the twelfth and final version studied. The maximum DIT for JasperReports remained at 5 in the entire 12 versions. Figure 5.3 shows the frequencies of number of classes at each DIT level in every fifth and final version. The number of classes at DIT one has the highest growth rate. This trend was also observed in a previous study by the same authors (Nasseri et al. 2008). The number of classes at DIT two and three also increased, but at a slower rate. Interestingly, the number of classes at DIT four and five stayed static at 10 and 5, respectively throughout.

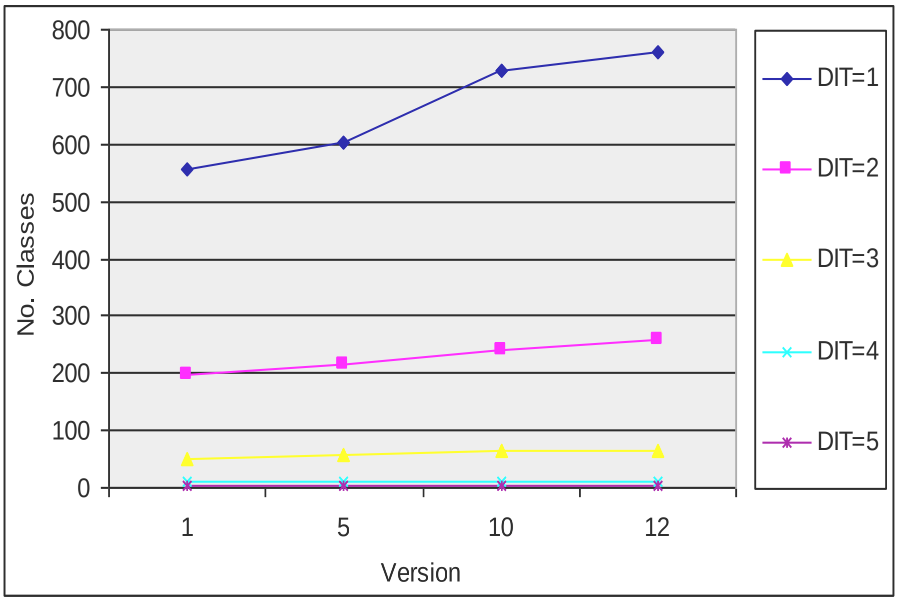


Figure 5.3. DIT frequencies in JasperReports (4 versions)

Part of the JasperReports system thus showed no changes (level four and five of hierarchy) despite the fact that other parts of the system were undergoing frequent change. This points to the possibility that classes at deep levels are not usually the focus of developer attention and often ignored. However, the lack of classes at those levels may mean that they remain relatively untouched by the developers anyway. Table 5.3 shows the evolution of classes across inheritance hierarchy between versions 1 and 5 in the style of Tables 5.1 and 5.2.

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	4	1	0	0	8	59
DIT2	0	•	0	0	0	2	17
DIT3	0	0	•	0	0	0	3
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

Table 5.3. Class evolution between version 1 to 5 in JasperReports

In the transition from version 1 to 5 of JasperReports, we see that only 5 classes were moved within the inheritance hierarchy. Visual inspection revealed that 2 new classes were added at DIT one; 3 classes from DIT one were moved as subclasses of one of the 2 new classes and one class from DIT one was moved as subclass of the second new class. Only 1 class was moved from DIT one to DIT three. The maximum DIT of that particular hierarchy reached 3 as a result. Table 5.4 shows the profile for JasperReports between versions 5 and 10.

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	2	0	0	0	5	134
DIT2	0	•	0	0	0	0	23
DIT3	0	0	•	0	0	0	7
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

Table 5.4. Class evolution between version 5 to 10 in JasperReports

In the transition from version 5 to 10, only 2 classes were moved from DIT one to DIT two. The majority of changes in the system occurred at higher levels of hierarchy (levels one and two). Table 5.5 shows the profile for JasperReports between version 10 and 12.

	DIT1	DIT2	DIT3	DIT4	DIT5	RemC.	NewC.
DIT1	•	12	0	0	0	1	45
DIT2	0	•	1	0	0	0	6
DIT3	0	1	•	0	0	0	2
DIT4	0	0	0	•	0	0	0
DIT5	0	0	0	0	•	0	0

Table 5.5. Class evolution between version 10 to 12 in JasperReports

From Table 5.5, 12 (i.e., 1.15%) of the total number of classes in the system were moved from DIT one to DIT two. From our inspection, we found that 3 new classes had been added at DIT one. 7 existing classes from DIT one were moved as subclasses of just a single, newly added class, 3 classes from DIT one and 1 class from DIT three were positioned as subclasses of the second newly added class. From the above analysis, we see that many of the changes essentially revolve around the 3 newly added classes. This localisation of change suggests that class movements might happen in clusters. Figure 5.4 shows the profile for changes in JasperReports.

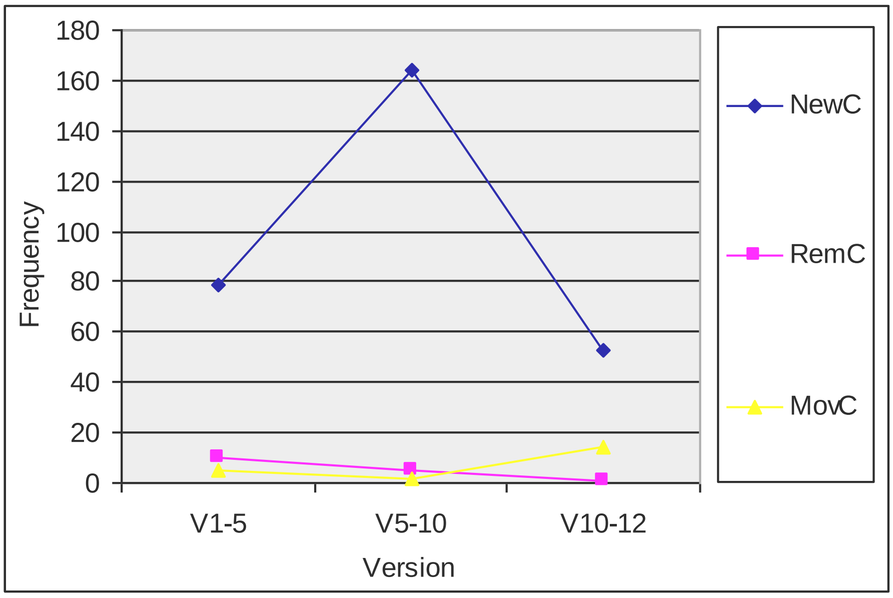


Figure 5.4. Changes in JasperReports

From Figure 5.4, we see that the majority of changes are addition of new classes in JasperReports (a total of 296). The total number of removed classes was 16 and the total number of moved classes within the existing inheritance hierarchy was 21. Considering the large number of classes in JasperReports (818 in the first version and 1098 in the twelfth version) we believe the inheritance hierarchy in the system was relatively stable (in terms of movement of classes within the hierarchy). We found no activity at DIT level four and five in the studied versions of JasperReports. This trend was also found for the same system in previous studies (Nasseri and Counsell 2009a, Nasseri and Counsell 2009b), where no activity in terms of number of *methods* and *attributes* was found at either level four or five.

SwingWT system started with 50 classes and contained 620 classes by version 22. The maximum DIT for SwingWT reached 7. Figure 5.5 shows the frequencies of classes at DIT one to three and Figure 5.6 shows the same trend for DIT 4 to 7 in every fifth and final version of the system. For the early versions of SwingWT, the maximum DIT was 3 and that gradually grew to 7. Figure 5.5 again shows that the majority of classes were added at DIT one.

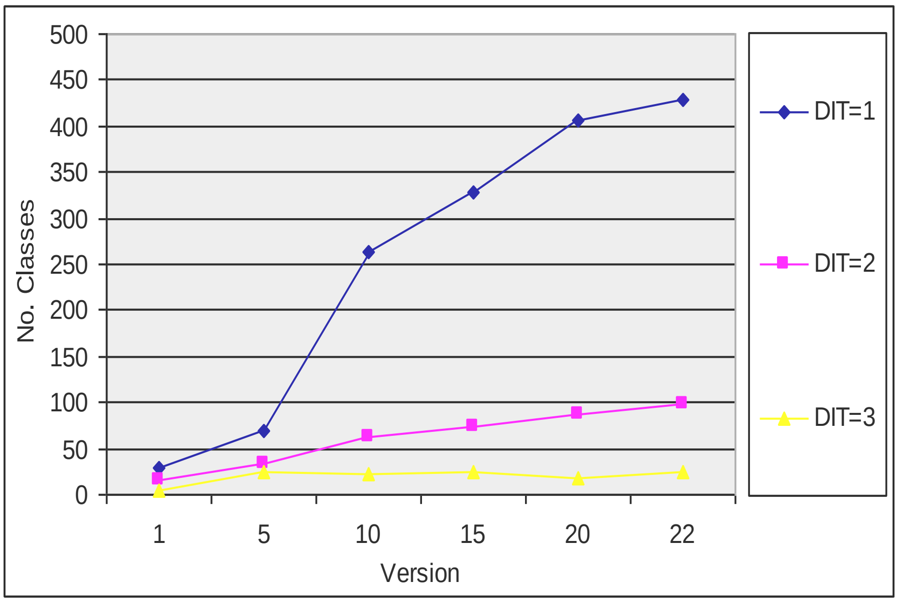


Figure 5.5. DIT 1, 2 and 3 frequencies in SwingWT (6 versions)

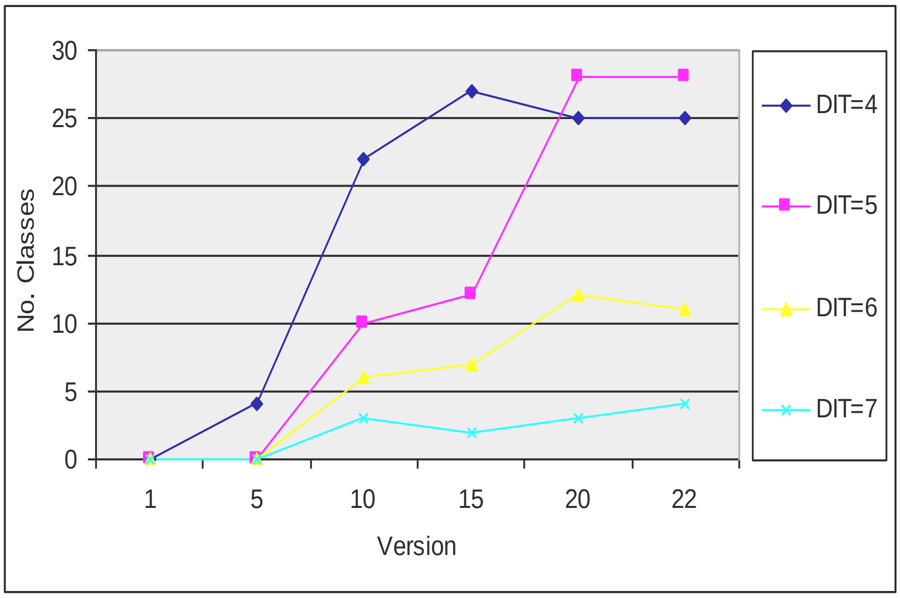


Figure 5.6. DIT 4, 5, 6 and 7 frequencies in SwingWT (6 versions)

Table 5.6 shows the movement of classes within the inheritance hierarchy between versions 1 and 5 of SwingWT.

	DIT1	**DIT2**	**DIT3**	**DIT4**	**DIT5**	**DIT6**	**DIT7**	**RemC.**	**NewC.**
DIT1	•	0	0	0	0	0	0	1	42
DIT2	0	•	0	0	0	0	0	1	17
DIT3	0	1	•	0	0	0	0	0	20
DIT4	0	0	0	•	0	0	0	0	4
DIT5	0	0	0	0	•	0	0	0	0
DIT6	0	0	0	0	0	•	0	0	0
DIT7	0	0	0	0	0	0	•	0	0

Table 5.6. Class evolution between version 1 to 5 in SwingWT

Only 1 class was moved up from DIT three to DIT two. From our inspection, we observed that the two ancestor classes of that particular class were removed and 1 new class added above (this changed its DIT from 3 to 2). In the transition from version 1 to 5, we see that the system grew considerably. Overall, 83 classes were added and only 2 classes were removed. Table 5.7 shows the profile for SwingWT in the transition from version 5 to 10.

	DIT1	**DIT2**	**DIT3**	**DIT4**	**DIT5**	**DIT6**	**DIT7**	**DelC.**	**NewC.**
DIT1	◆	0	0	1	0	0	0	2	196
DIT2	0	◆	10	3	3	0	0	0	45
DIT3	0	1	◆	11	4	6	0	0	11
DIT4	0	0	0	◆	1	0	2	0	5
DIT5	0	0	0	0	◆	0	0	0	2
DIT6	0	0	0	0	0	◆	0	0	0
DIT7	0	0	0	0	0	0	◆	0	1

Table 5.7. Class evolution between version 5 to 10 in SwingWT

We see that, 42 classes (i.e., 31.82% of all classes) were re-located across the hierarchy. Only 1 class was moved from DIT one to DIT four. A new class was added above that class at DIT one (the name of the class was JSWMenuComponent and its new superclass was named AbstractButton); the same class and its new superclass were placed as subclasses of an existing class at DIT two (JComponent). Moreover, 10 classes were moved from DIT two to DIT three. It was revealing that in version 5, those 10 classes were sibling classes of the JComponent class at DIT two which in version 10 were then placed as subclasses of JComponent. Three classes were moved from DIT two to DIT four. Those three classes were subclasses of the Component class which were in turn moved as subclasses of the newly added class (AbstractButton) at DIT three. Three classes were moved from DIT two to DIT five. The 3 classes were subclasses of the JSWMenuComponent class at DIT one for which a superclass was added (AbstractButton) and was moved to DIT four.

All movements of classes therefore revolved around only four classes (Component, at DIT one, itself superclass of JComponent, AbstractButton and JSWMenuComponent). This suggests that the system was designed in such a way that the classes in one part of the system were highly amenable to easy movement which, in practice, could reflect a portable design. In addition, we found that some classes were moved, for instance, directly from DIT one to DIT four; as a result, its subclasses were correspondingly moved from DIT two to DIT five. A dependency between groups of classes seems to exist. Table 5.8 shows the profile for SwingWT between versions 10 and 15.

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	RemC.	NewC.
DIT1	◆	2	3	0	0	0	0	23	90
DIT2	0	◆	0	2	0	0	0	6	15
DIT3	1	0	◆	3	0	0	0	0	3
DIT4	0	1	0	◆	2	0	0	3	6
DIT5	3	0	0	0	◆	0	0	0	1
DIT6	0	0	0	0	0	◆	0	0	1
DIT7	0	0	0	0	2	0	◆	0	1

Table 5.8. Class evolution between version 10 to 15 in SwingWT

Between version 10 and 15, 19 classes were moved within the inheritance hierarchy. A noticeable feature of Table 5.8 is the number of classes that moved from DIT five to DIT one. Classes may move one or two levels as a result of addition/deletion of a class in the hierarchy. However, movement of classes from *root* (below Object) closer to the *leaf* of the hierarchy or vice versa implies a haphazardly structured hierarchy. Our analysis revealed that only one new class was added at DIT three and two classes were moved as subclasses of that class further reinforcing the view that 'mutually dependent' classes do tend to move in clusters. Nine (i.e., 47.37%) of the 19 moved classes were those classes moved within the hierarchy in the previous transition (between version 5 and 10). In other words, a large subset of the 19 moved classes between version 5 and 10 were moved *again* between version 10 and 15. This was an interesting feature to emerge from our analysis, suggesting that a subset of classes were prone to movement. SwingWT is a GUI application which uses inheritance extensively. We believe the deep level of inheritance makes it harder for a developer to move classes within the hierarchy. Movement of one class may require several classes to be moved due to their superclass and subclass relationships which tend to be strongly tied. For example, if a class is moved from DIT two to DIT four, any subclasses moved with that class changes its DIT from three to five; we found ample evidence of this in SwingWT.

In the transition from version 10 to 15, 32 classes were removed from the system, 19 (i.e., 59.38%) of which were inner classes suggesting that inner classes can easily be deleted from the system. This was also a revealing feature of our analysis. Inner classes might be easier to remove from a system because they are encapsulated within their outer enclosing classes. As such, inner classes have no dependencies (i.e. coupling) with other classes other than that imposed by the enclosing class. Table 5.9 shows the number of classes moved around the inheritance hierarchy in the transition from version 15 and 20.

	DIT1	DIT2	DIT3	DIT4	DIT5	DIT6	DIT7	RemC.	NewC.
DIT1	◆	1	0	0	8	0	0	12	96
DIT2	0	◆	1	0	0	2	0	2	15
DIT3	0	1	◆	12	0	1	1	0	1
DIT4	2	1	4	◆	14	1	0	1	3
DIT5	0	0	1	6	◆	5	0	0	3
DIT6	0	1	1	0	3	◆	2	0	1
DIT7	0	0	0	0	0	2	◆	0	0

Table 5.9. Class evolution between version 15 to 20 in SwingWT

Seventy classes (i.e., 14.37% of the total) changed their position in the hierarchy. Only one class was moved from DIT one to DIT two. One class was added at DIT one and the moved class was placed as a subclass of that new class. Similarly, one class was added at DIT four (in the most change-prone part of the system) and 8 classes moved to become subclasses of that new class. We also see that 12 classes were moved from DIT three to DIT four. Those 12 classes were subclasses of a single class moved from DIT two to DIT three (its subclasses were moved from DIT three to DIT four). Likewise, one of the 12 classes which were moved from DIT three to DIT four had 14 subclasses - all of which were also moved from DIT four to DIT five.

In the transition from version 15 and 20, we again found that the vast majority of movement of classes took place due primarily to the movement of their superclasses. Furthermore, we found that 14 (i.e., 20%) of the 70 moved classes were those repositioned within the hierarchy in the previous transition (i.e., between version 10 and 15); 38 (i.e., 54.29%) of the 70 moved classes were those classes which were repositioned in the transition between version 5 to 10. We also found the same 9 classes to be moved in every transition from version 5 to 20, supporting the view that there are certain subsets of classes so tightly coupled that they cannot be decomposed; they *need* to be moved around together (even though the functionality of *all* nine classes might not be required where they are moved). These classes would be ideal candidate classes for re-engineering or refactoring.

From Table 5.9, 119 new classes were added, 11 of which (i.e., 9.24%) were those classes removed from the system in the previous transition between versions 10 and 15. In addition, we found that between versions 15 and 20, 15 classes were removed from the system all of which again were inner classes. Anecdotal evidence would suggest that the existence of constructs such as Java inner classes influences the scrutinizing role of the developer by complicating the task of maintenance. Inner classes allow a nested class access to the attributes of the enclosing class and have been the subject of certain criticism since they add a level of complexity to the system (McGraw and Felten 1998, Sintes 2001).

In the transition from version 20 and 22, only 1 class was moved from DIT six to DIT seven (this class was also moved from DIT five to DIT six between version 15 to 20). 1 inner class was removed and, overall, 42 new classes were added of which 24 were added at DIT one, 12 classes at DIT two and 6 classes at DIT three. In SwingWT, the total number of moved classes was 133 of which only 6 (i.e., 4.51%) classes were inner classes. The total number of new classes in the system was 621 of which 141 (i.e., 22.71%) were inner classes, and the total number of removed classes was 52 of which 37 (i.e., 71.15%) were inner classes suggesting again that inner classes in SwingWT are far more amenable to deletion than 'regular' 'unenclosed' classes. Figure 5.7 shows the overall changes in SwingWT in the same format as Figures 5.2 and 5.4.

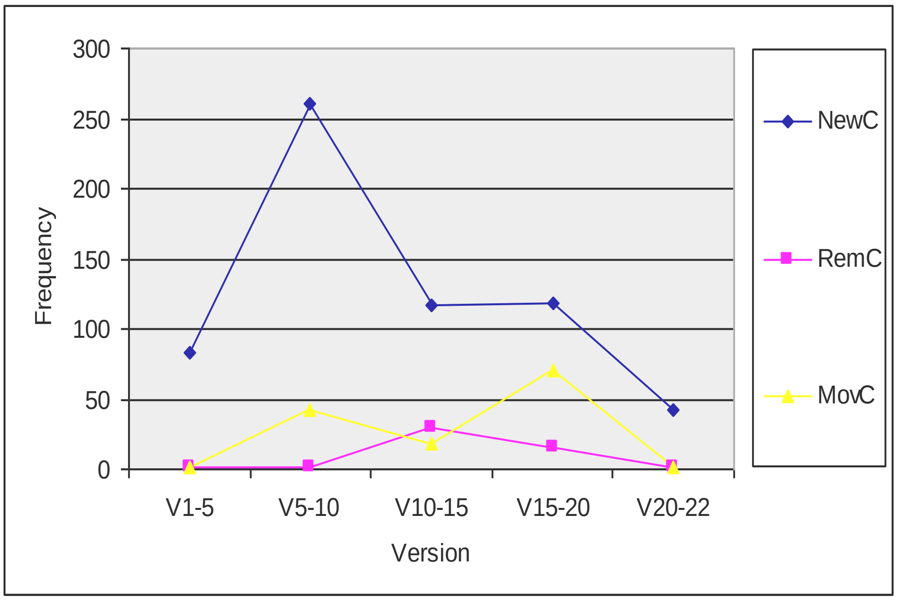


Figure 5.7. Changes in SwingWT

From Figure 5.7, we see that the majority of changes are increases in number of classes. Two peaks are visible in terms of number of moved classes between version 5 to 10 and 15 to 20. The trends in number of added, removed and moved classes contrast. Each seems to have a peak at different stages of evolution.

The Tyrant system consisted of 122 classes in version 1 and contained 273 classes in version 45. The maximum DIT for Tyrant was 5. Figure 5.8 shows the frequencies of number of classes at each DIT level in every fifth and final version of Tyrant. An interesting feature is that in version 4, the system underwent a major change. The maximum DIT dropped from 5 in version 4 to 3 in version 5. The number of classes at DIT one increased from 45 in version 4 to 96 in version 5. The number of classes at DIT two dropped from 42 to 13. Finally, the number of classes at DIT three increased from 22 in version 4 to 63 in version 5. Since we found that the system went through significant changes between version 4 and 5, we therefore included that transition in our analysis.

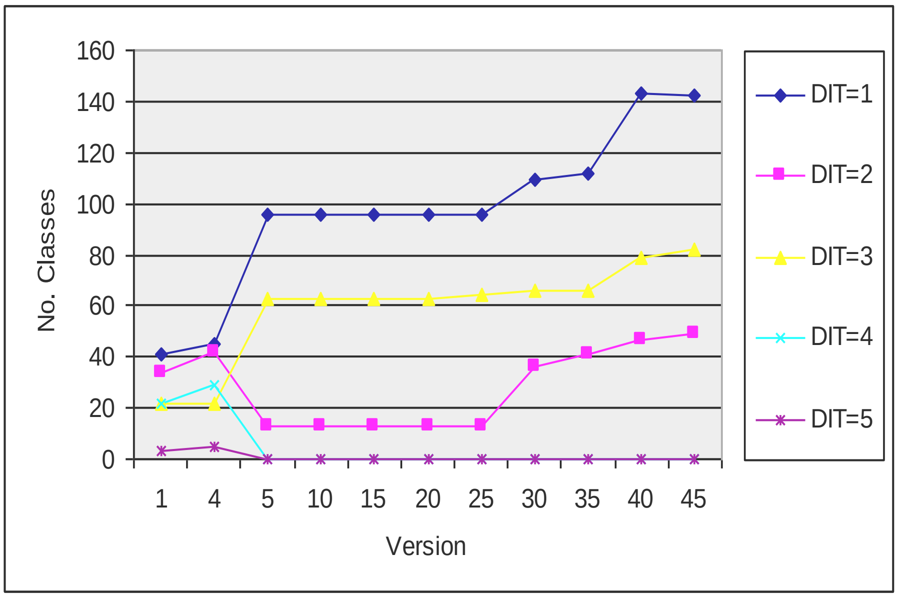


Figure 5.8. DIT frequencies in Tyrant (11 versions)

Between version 1 and 4, the total number of classes in the system increased from 122 to 143. In the transition from version 1 to 4, only five classes were relocated in the hierarchy. A single class was added at DIT one and two existing classes from DIT one were placed as its subclasses. In addition, 2 new classes were added at DIT three and 2 classes from DIT three were moved as subclasses of one of them. One class from DIT three was moved to become a subclass of the other. We found that no classes were removed from the system. Table 5.10 shows the number of moved classes at each DIT level in Tyrant, as well as the number of added and removed classes between versions 4 and 5 in the same format as Table 5.9.

	DIT1	**DIT2**	**DIT3**	**RemC.**	**NewC.**
DIT1	◆	3	1	11	32
DIT2	12	◆	8	18	4
DIT3	11	1	◆	9	53
DIT4	11	1	0	17	0

Table 5.10. Class evolution between version 4 to 5 in Tyrant

From Table 5.10, 48 classes were moved within the inheritance hierarchy. The maximum DIT dropped correspondingly from 5 to 3. The total number of classes at DIT five in version 4 was 5, all of which were then removed from the system. The total number of classes at DIT four was 29, 11 of which were moved up to DIT one. The majority of classes (34 classes in total) were moved to DIT one. We also found that 56 classes were repositioned across the hierarchy in the entire 45 versions of Tyrant, 48 (i.e., 85.71%) of which were moved in the transition from version 4 to 5. The total number of new classes was 226, 110 (i.e., 48.67%) of which were added between version 1, 4 and 5. The total number of removed classes was 75, 60 (i.e., 80%) of which were removed in that same transition (version 4 to 5). A major re-engineering initiative seems to have occurred between versions 4 and 5. Figure 5.9 shows the trend in change frequencies in Tyrant.

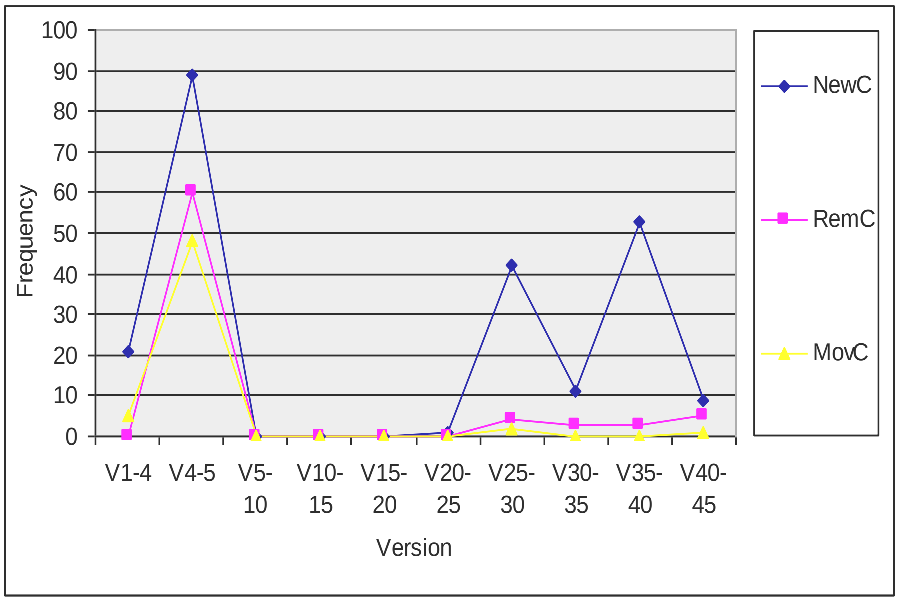


Figure 5.9. Changes in Tyrant

Following the re-engineering of the system and flattening of the hierarchy, the inheritance hierarchy in Tyrant stabilized in terms of movement of classes. Only 2 classes were moved from DIT one to DIT two between versions 25 and 30, and only 1 further class was moved from DIT two to DIT one between versions 40 and 45. The remaining changes were all either increases or decreases in the number of classes.

5.4.2 Analysis of class characteristics

So far we have investigated the trends in movement of classes within the existing inheritance hierarchy. The decision to move a class within the hierarchy may be influenced by the characteristics of specific classes. For example, larger classes may be more frequently moved, as they contain more functionality and their relocation may have a significant impact on system comprehension. Similarly, tightly coupled classes may be more frequently moved within the hierarchy to reduce class coupling and improve system comprehension. We therefore speculate that (i) larger classes are more likely to be moved within the hierarchy than smaller classes, and (ii) tightly coupled classes are more likely to be moved within the hierarchy than loosely coupled classes. We analysed characteristics of the moved and static classes in all four systems to determine whether this was actually the case in the four studied systems. We analyzed two features of the classes in the four systems (i) class size, given by number of methods (NOM) metric of Lorenz and Kidd (Lorenz and Kidd 1994) and (ii) coupling, given by the Message Passing Coupling (MPC) metric of Li and Henry (Li and Henry 1993). We formed and tested the following hypotheses in order to investigate whether larger classes and tightly coupled classes were more frequently moved within the hierarchy.

The null hypothesis H01 states: Class movement and relocation is not influenced by class size.

An alternative hypothesis HA1 states: Larger classes, given by their NOM, are more likely to be moved within the hierarchy than smaller classes, as a system evolves.

The null hypothesis H02 states: Class movement and relocation is not influenced by class coupling.

An alternative hypothesis HA2 states: Tightly coupled classes, given by their MPC, are more likely to be moved within the hierarchy than loosely coupled classes, as a system evolves.

Table 5.11 shows the maximum (Max.), mean, median and standard deviation (STDEV) of NOM and MPC for moved (prior to their movement) and un-moved classes in the four studied systems.

		Moved Classes				Un-moved Classes			
		Max.	Mean	Median	STDEV	Max.	Mean	Median	STDEV
HSQLDB	NOM	14	7.25	7	4.95	171	12.57	7	17.90
	MPC	73	23.37	9.5	30.08	593	42.32	16	73.04
Jasper Reports	NOM	49	12.67	15	11.05	166	9.84	4	19.41
	MPC	74	7.76	0	19.08	1238	23.15	5	69.29
SwingWT	NOM	102	14.33	9	15.93	170	9.09	4	13.86
	MPC	224	21.11	10	33.68	279	11.16	0	26.14
Tyrant	NOM	88	12.96	9	14.65	63	6.27	2	9.84
	MPC	151	38.62	25	42.63	268	29.86	13	45.08

Table 5.11. Class characteristics in the four systems

From HSQLDB, all values (with the exception of the median NOM) of NOM and MPC for moved classes are relatively smaller than their corresponding values for un-moved classes, suggesting that classes with relatively low NOM and MPC values did tend to be favoured when moving classes across the inheritance hierarchy. The mean and median values of NOM for moved classes in JasperReports show a different trend to that of HSQLDB. If we consider the mean values in JasperReports, we see that classes with higher NOM values were moved within the inheritance hierarchy. We see that all MPC values for moved classes in JasperReports are smaller than their corresponding values for un-moved classes. This again implies that in JasperReports, classes with fewer coupling features were moved within the hierarchy. In SwingWT, we see that all values of NOM and MPC, with the exception of their Max. values for moved classes are larger than their corresponding values for un-moved classes, suggesting that larger classes, given by NOM, and highly coupled classes, given by MPC, were more frequently moved within the system inheritance hierarchy. Furthermore, all values of NOM and MPC for moved classes in Tyrant, with the exception of Max. and STDEV MPC, seem to be higher than their corresponding values for un-moved classes. This again implies that in Tyrant, larger classes and relatively highly coupled classes were moved across the hierarchy.

To formally test the null hypotheses (H01 and H02), we carried out two, one-tailed non-parametric Mann-Whitney U-tests (Hinkle et al. 1995) on moved and un-moved classes and their NOM and MPC. Table 5.12 shows the results of the Mann-Whitney U-test carried out on moved and un-moved classes and NOM in the four systems.

Classes	N	Mean-Rank	Sum of Rank	M-Whitney-U	p-Value	Z-Score
Moved	220	1472.28	323902.5	174727.5	0.000	-6.474
Un-Moved	2156	1159.54	2499973.5	◆	◆	◆

Table 5.12. Mann-Whitney U-test for moved/un-moved classes and NOM (all systems)

From Table 5.12, there is a significant difference between the two samples (moved and un-moved classes) in the four systems. The mean rank value for moved classes is higher than that of un-moved classes, suggesting that larger classes, given by NOM, were more frequently moved within the hierarchy than smaller classes. The p-value for the test is < 0.01 i.e., the test is statistically significant at the 1% level. Table 5.13 shows the results of the Mann-Whitney U-test carried out on moved and un-moved classes and MPC.

Variables	N	Mean-Rank	Sum of Rank	M-Whitney-U	p-Value	Z-Score
Moved	220	1305.29	287163.5	211466.5	0.0035	-2.679
Un-Moved	2156	1176.58	2536712.5	◆	◆	◆

Table 5.13. Mann-Whitney U-test for moved/un-moved classes and MPC (all systems)

From Table 5.13, we again see a significant difference between the two samples. The mean rank value for moved classes is relatively higher than that of un-moved classes in the four systems, suggesting that classes with higher coupling, given by MPC, were more frequently moved within the hierarchy than classes with lower coupling. The p-value for the test is < 0.01 i.e., the test is statistically significant at the 1% level.

Based on the evidence in Table 5.12, we see that in the four systems, larger classes were more frequently moved within the hierarchy than smaller classes - we are therefore in the position to reject the H01in favour of HA1; larger classes are more likely to be moved within the hierarchy than smaller classes, as a system evolves. In terms of coupling (Table 5.13), classes with high MPC were moved more frequently than classes with low MPC - we are therefore in the position to reject the H02 in favour of HA2; tightly coupled classes are more likely to be moved within the hierarchy than loosely coupled classes, as a system evolves.

One possible explanation for movement of larger and highly coupled classes may be the lack of cohesion in those classes. Smaller classes and loosely coupled classes may be more cohesive than larger classes and tightly coupled classes respectively - they therefore often remain un-moved because they have not deteriorated sufficiently to necessitate being moved. To investigate this feature, we conducted two widely used non-parametric cross-correlations (Kendall’s and Spearman’s) of size (given by NOM), coupling (given by MPC) and cohesion (given by LCOM metric of Chidamber and Kemerer (Chidamber and

Kemerer 1994)). Table 5.14 shows the correlation values of NOM versus LCOM and MPC versus LCOM in the four systems.

	Kendall’s	Spearman’s
NOM vs. LCOM	0.246**	0.310**
MPC vs. LCOM	0.255**	0.333**

Table 5.14. Correlations of NOM/MPC and LCOM

From Table 5.14, all the values are double asterisked indicating that the correlations are significant at the 1% level. There is a strong and significant relationship between NOM and LCOM, and MPC and LCOM in the four systems. This result implies that as the size and/or coupling of a class increases, its cohesion decreases. We believe that the key motivation for frequent movement of larger classes and tightly coupled classes within the hierarchy is the lack of cohesion in those classes.

Class movement and re-location provides valuable information to software managers and developers. For example, in this chapter we have found that larger classes and highly coupled classes are more frequently moved within the hierarchy. Scrutiny of the data also revealed that larger classes and highly coupled classes were less cohesive which is why they might be re-located more frequently. Software managers and/or developers can reduce class size by extracting new classes and reducing coupling in favour of cohesion to improve structural stability of their systems. In addition to that, class movement and re-location is an activity often undertaken during the process of re-engineering. Our empirical evidence for one of the systems (Tyrant) showed that post re-engineering of the system (between versions 4 and 5), the system showed a smooth evolution with minimal structural changes. Class movement and re-location can also have positive implications on system evolution. In that respect, we believe class movement and re-location can have a significant role in the structural stability of a system.

Movement of classes with higher NOM and those with higher MPC can also be justified on the basis that developers tend to reduce structural complexity by moving large and tightly coupled classes. Table 5.15 shows the maximum (max.) mean, median, and standard deviation (STDEV) values for classes after they were moved. From Table 5.15, all values of NOM and MPC for HSQLDB, SwingWT and Tyrant (in Tyrant with the exception of Median NOM) are considerably higher than their respective values presented in Table 5.11 (in Table 5.11, the same data for moved classes prior to their movement is presented), suggesting a significant growth in terms of NOM and MPC in the systems. For JasperReports however, the opposite has occurred and all values of NOM and MPC for moved classes are smaller than their corresponding values presented in Table 5.11. This was surprising considering the significant growth of the system (818 classes in version 1 and 1098 classes by version 12).

	Moved classes after their re-location				
		Max.	Mean	Median	STDEV
HSQLDB	NOM	134	26.63	9	44.64
	MPC	498	93	13	171.30
JasperReports	NOM	48	6.38	4	10.10
	MPC	23	3.62	0	7.62
SwingWT	NOM	123	20.54	17	19.39
	MPC	278	32.67	19	44.77
Tyrant	MPC	127	14.30	8	21.70
	NOM	365	65.36	39.5	71.02

Table 5.15. Descriptive statistics for the moved classes after their movement

Given the results that larger classes and highly coupled classes, due possibly to their lack of cohesion, were more frequently moved within the inheritance hierarchy, we speculate that class movement may actually improve class cohesion. That is, class movement may have positive implications for class cohesion. To investigate this phenomenon we formed and tested the following hypotheses.

Null hypothesis H03 states: Class cohesion is not influenced by class movement and relocation.

Alternative hypothesis HA3 states: Class movement and relocation improves class cohesion.

To formally test the null hypothesis (H03), we categorised moved classes into two groups, 1) before their movement and 2) after their movement. We took class movement as the main unit of our analysis and conducted a Wilcoxon signed-rank test (Wilcoxon 1945) to identify the impact of class movement on class cohesion. Table 5.16 shows the results of the Wilcoxon signed-rank test carried out on moved classes before/after their movement and their cohesion.

		N	Mean Rank	Sum of Ranks	Z-Score	P-Value
After - Before	Negative Ranks	46[a]	70.46	3241	-0.873[a]	0.383
•	Positive Ranks	73[b]	53.41	3899	•	•
•	Ties	101[c]	•	•	•	•
•	Total	220	•	•	•	•

a After < Before
b After > Before
c After = Before

Table 5.16. Wilcoxon signed-rank test for moved classes before/after their movement

From Table 5.16, we see that the 'After' variable appears first suggesting that it was first entered into the equation. The negative ranks shows that 46 ranks of 'Before' were greater than 'After'. The positive ranks shows that 73 ranks of 'Before' were smaller than 'After'. Finally, the Ties ranks shows that 101 ranks of 'Before' and 'After' were equal. The z-score and p-value are -0.873 and 0.383 respectively, implying that the test is not statistically significant. We are therefore in a position to conclude that there is not a significant difference between the cohesion of classes before and after their movement. We accept H03: Class cohesion is not influenced by class movement and re-location.

One tenet of software evolution states that as a software system evolves, its size, coupling and complexity increases and cohesion of the system deteriorates. The systems studied herein grew significantly (i.e., HSQLDB started with 56 classes and ended with 358 classes; JasperReports started with 818 classes and comprised 1098 classes by the final version; SwingWT started with 50 classes and comprised 620 classes by the final version and, finally, Tyrant started with 122 classes and ended with 273 classes). In that respect, the evidence presented in Table 5.16 may imply that class cohesion has not been improved by class movement and re-location. On other hand, the evidence also indicates that class cohesion has not deteriorated (there isn't significant difference between the cohesion of the classes before and after their movement), despite the fact that the systems grew significantly.

5.5 Discussion

Class movement and relocation is the process of restructuring a system to improve its comprehensibility and ease maintenance efforts in future releases of a system. Our findings can be of importance to software managers to predict how OSS change over time, which classes are more frequently moved within an inheritance hierarchy, how to minimise those movement, and most importantly, target refactoring efforts on regularly changing parts of a system. In this chapter, a number of points were observed that might be of interest to software practitioners. Firstly, we found that in one of the systems (SwingWT), the majority of the changes were made in one specific part of the system. A number of classes changed their DIT values on a regular basis which could be considered poor practice. Additionally, in the same system we found that 11 classes, removed from the system in one version, were re-integrated into the system in subsequent versions. Furthermore, in one of the systems (Tyrant) in the transition from version 4 to 5, we observed that the system underwent major changes as a result of which the maximum DIT dropped from 5 to 3. We believe developers of the system restructured the inheritance hierarchy and, consequently, the hierarchy in the system stabilized in terms of class movement (between versions 5 and 45, only 3 classes moved within the hierarchy). We believe that system re-engineering could help avoid constant movement of classes within the hierarchy; early re-engineering can help impede on-going structural change in a system.

In addition, we observed that developers of OSS tend to opt for three levels of inheritance rather than deeper levels. We found evidence of ‘collapsing’ hierarchies (to bring the classes up to shallow levels). One of the systems (JasperReports) where five levels of inheritance was used, no activity was found beyond level three. Classes at DIT four and five showed no changes in the entire set of 12 versions of the system. This trend was also found in (Nasseri and Counsell 2009a, Nasseri and Counsell 2009b) where levels four and five of hierarchy in JasperReports showed no changes in terms of increases or decreases in number of methods and attributes. Finally, in one of the systems (SwingWT) where the maximum length of hierarchy reached 7, the hierarchy tended to remain relatively unstable, i.e., the structure of the hierarchy tended to change constantly. A logical way of using inheritance would be to take the advantage of inheritance yet avoid its inherent complexity. This seems to be achieved only when using inheritance at shallow levels and we therefore posit that care should be taken when using inheritance beyond level three.

5.6 Summary

In this chapter, we presented an empirical study of evolution of classes in four Java OSS. Inheritance data was collected using the JHawk tool (described in Section 2.3.2.6) and changes observed in terms of newly added classes, deleted classes and moved classes within the inheritance hierarchies in multiple versions of the four systems. We suggest that if developers are restricted to changes in just one part of the system, then that reflects a poor design and/or poorly applied previous maintenance in that specific part of the system. In theory, changes should be evenly spread across all parts of the system, but in practice there seem to be ‘hot spots’ in a system (i.e., areas of code that require constant developer attention). Identifying where these areas tend to occur and, more importantly, why they occur, could help future effort to be directed and estimated.

We also showed that a maximum of three levels of inheritance may be more desirable than deeper levels. Developers of the systems studied tended to focus most of their activity (from the change data) at, and above, level three rather than below level three. We also found very little activity below level three of the inheritance hierarchies in the studied systems (with the exception of SwingWT where seven levels of inheritance were used). In addition, we found evidence of hierarchies being ‘collapsed’ to bring classes up to shallower levels rather than them remaining at deep levels. Interestingly, we also found evidence to suggest that in the set of OSS studied larger classes and tightly coupled classes were more frequently moved within the hierarchy than smaller classes and loosely coupled classes. We investigated the reasons why this may have occurred and found smaller classes and loosely coupled classes to be more cohesive than larger classes and tightly coupled classes; we believe the lack of cohesion in those classes may have been a factor that may have influenced the decision to move those classes. We believe these results are of some relevance, since software systems spend most of their `life' being maintained. Understanding where this change tends to take place helps predict future maintenance activity and target scarce refactoring resources to areas where most benefit will be achieved.

CHAPTER 6 Inheritance and Method Invocation

6.1 Introduction

In the previous chapter we investigated the trends in changes of inheritance in terms of addition, deletion and movement of classes within and across inheritance hierarchy. Exploring the evolution of inheritance hierarchy in a system can provide valuable insights into the system dynamics from an inheritance perspective. A recent study of seven Java OSS by the authors (Nasseri et al. 2008) showed that approximately 96% of incremental, evolutionary changes were made to classes at levels one and two of the inheritance hierarchy. Only 4% of the same changes were made to classes at, and beyond, level three. The majority of system functionality was found at levels one and two. One conclusion that we can draw from the trend is that if levels one and two are where the bulk of the functionality exists, then that is where classes will tend to invoke functionality, even to classes outside the line of superclasses to the root. While we thus know how inheritance hierarchies change over time, what is not so obvious is how classes in an inheritance hierarchy interact with one another and how their interactions evolve, as opposed to that of the system as a whole.

In this chapter, we investigate the evolution of inheritance from a method invocation perspective and its impact on class cohesion in four Java OSS. We distinguish between method invocations within the line of classes to the root of the hierarchy and 'external' method calls to classes for which there is no direct line of superclasses to the root; in other words, to what extent do classes invoke methods in classes 'across' the hierarchy rather than 'up' it or 'down' it. To further investigate this phenomenon, inheritance, size, cohesion and method invocation data was extracted using the JHawk tool (JHawk 2008). The main research question that we sought to explore was: *in light of the 'top heavy' nature of Java hierarchies shown in a previous study by the authors* (Nasseri et al. 2008), *to what extent do classes take advantage or otherwise of superclass functionality (i.e., the subclass-superclass relationships inherent in every inheritance hierarchy) when invoking functionality of other classes?*

In the next section we present the motivation for the study. Section 6.3 describes the design of the empirical study including the four OSS studied, data collected and the methodology adopted. We then present summary data (Section 6.4). In Section 6.5 we present the data analysis in single and multiple versions of the systems. Section 6.6 presents a discussion of the validity of the study and the generalisation of the results; finally, we provide the summary and conclusions of the empirical study in Section 6.7.

We note that a part of this chapter has been submitted for publication in (Nasseri and Counsell 2009c)

6.2 Study Motivation

The chief motivation for the study in this chapter arises from the lack of empirical studies into how classes within an inheritance hierarchy interact with one another and how that interaction evolves as the corresponding systems evolve. Such studies can bring to light patterns that may exist in system evolution which, in turn, may help software project managers and developers prevent the decay of code in future versions. While we can view system evolution at a class level relatively easily and hence view systems as a collection of connected black boxes, such studies hide the lower-level granularity of functionality and extent of coupling a class has with other classes. Trends in such behaviour, particularly if it crosses the width rather than depth of an inheritance hierarchy, can create problems for developers, since it requires them to follow links that we normally associate with ‘spaghetti code’.

6.3 Empirical Study Design

The study in this chapter reports the results of a quantitative analysis of class interaction through method invocation in four Java OSS. The analysis takes into account class interaction through association, when a class access methods of another class in the system outside the direct line of classes to the root, and class interaction within a hierarchy, when a class accesses methods of its superclasses.

6.3.1 The four open-source systems

The four OSS on which our study is based were HSQLDB, JasperReports, SwingWT and Tyrant (see systems 1, 3, 5 and 8 described in Section 2.3.2.5). For this empirical study, we included all available versions of the four systems.

6.3.2 Data Collected

For this study we used JHawk (JHawk 2008), described in Section 2.3.2.6, to extract the following OO metrics:

1. Depth of Inheritance Tree (DIT).
2. Number of Methods (NOM).
3. Number of calls to methods within Hierarchy (HIER).
4. Number of External method calls (EXT).
5. Lack of Cohesion Of the Methods in a class (LCOM).

(see metrics 1, 5, 7, 8 and 9 described in Section 2.3.2.6).

6.3.3 Methodology

In the following, we describe how the measures were used in the study.

- The DIT was used to identify and then measure the HIER and EXT metrics at each level of inheritance hierarchy.
- The NOM was used to measure class size in the systems studied.
- The HIER was used to measure the amount of interaction between classes within the inheritance hierarchy.
- EXT was used to measure the amount of interaction between classes other than those in the hierarchy.
- LCOM was used to measure the cohesion in classes of the four systems.

Following the extraction of the measures from each of the systems, we computed the total HIER and EXT per DIT level and observed how they changed at each DIT level as the systems evolved. We then analyzed the HIER and EXT values, taking into account the number of classes at each DIT level in the final versions of the four systems. We assessed the impact of HIER and EXT on class cohesion using the Mann-Whitney U-test (Hinkle et al. 1995); finally, we carried out a two-tailed, parametric correlation (Pearson's) and two non-parametric correlations (Kendall's and Spearman's) of NOM versus HIER and EXT in the final versions of the four systems to identify any relationship that may exist between the size of a class, given by NOM, and class coupling, given by HIER and EXT.

6.4 Summary Data

Table 6.1 provides summary statistics of changes of HIER and EXT values in the four systems. The data is in the form of maximum (Max.), minimum (Min.), median (Med.), mean (Mean) change of HIER and EXT. Table 6.1 also presents the variance (Var.) for the changes of HIER and EXT for the four systems. In addition, Table 6.1 shows the normalized (Norm.) percentage of 'Max.' for each measure indicating what percentage of number of HIER and EXT in initial versions of each system 'Max.' represents. For example, the maximum change of EXT in HSQLDB was 4069 which represented an increase of 238% over the total number of EXT in first version of that system. Most noticeable from Table 6.1 are the exceptionally high values of EXT compared with those of HIER for each system. While we accept that some level of coupling between classes measured by EXT is reasonable, the extent of that method invocation is exceptionally high.

Systems	Measures	Max.Ch	Norm.	Min.Ch	Med.Ch	Mean Ch	Var.
HSQLDB	HIER	84	N/A	0	11	23.2	1176.7
	EXT	4069	238%	6	1169	1552.8	2626167
Jasper Reports	HIER	17	4%	0	6	7.36	32.66
	EXT	1116	8%	75	222	388.55	116039.5
SwingWT	HIER	255	2833%	0	17	44.29	3653.61
	EXT	1864	471%	36	468	580.90	229445
Tyrant	HIER	200	97%	0	0	6.61	917.41
	EXT	3779	205%	0	28.5	188.86	349841.7

Table 6.1. Summary Change data for the HIER and EXT in the four systems

Since the first version of HSQLDB contained zero HIER, the Norm. HIER is therefore not applicable to this system. On a system-by-system basis, the highest mean change for EXT belongs to the HSQLDB system, indicating that significant maintenance has been made to the pattern of method interactions in this system. The low mean change values for HIER and EXT in Tyrant indicate that the system is relatively stable in comparison to its counterparts. In (Nasseri et al. 2008), it was noted that between version 4 and 5, Tyrant was subject to a major re-engineering effort through addition and deletion of large numbers of classes. As a result, the maximum DIT in the system dropped from five to three. Following that re-engineering effort, the system showed no change in terms of number of classes for a significant number of versions. This trend in Tyrant was also observed in two previous studies (Nasseri and Counsell 2009b, Nasseri and Counsell 2009a) where, after version 5, this system also showed no change in number of methods and attributes for a significant number of versions. In the context of the study described, this implies that apart from a significant effort at one stage in its lifetime, the Tyrant system was less susceptible to coupling across the hierarchy than the other three systems.

The high variance in values from Table 6.1 give the impression that each of the systems was subject to significant change at some point and those changes resulted in a correspondingly significant amount of external method calls given by EXT. Equally, the HIER measure reflects a lack of willingness of subclasses to use superclasses and vice versa. In the following sections, we present a more detailed description of the measures on each system.

6.5 Data Analysis

6.5.1 Method Invocation

We start our analysis with the HSQLDB system. The maximum DIT found for the HSQLDB system was 4. Figure 6.1 shows the frequencies of HIER at each DIT level in the six versions of HSQLDB. In order to examine whether the addition of new classes was the primary reason for the increase in number of HIER, we also show the number of classes with at least 1 HIER (Figure 6.2) at each DIT level across the same six versions. Classes are primarily calling the methods of their ancestors and since classes at DIT one have no superclasses, we therefore found all the HIER to occur at DIT level two and below which is why we excluded the DIT one from figures showing the evolution of HIER. From Figure 6.1, the majority of HIER exists at DIT level two and, to a lesser extent, level three. We see a sudden rise in HIER in the transition between versions 3 and 4. Between those versions, the number of classes at DIT one increased from 114 to 247, and the number of classes at DIT two and three increased from 29 and 4 to 63 and 12, respectively. In addition, in version 4 the maximum DIT increased to four and only 1 class was observed at this level, which was removed from the system in version 6. The lack of classes at DIT four explains the lack of HIER at that level. Figure 6.2 shows a similar trend to that of Figure 6.1. We see that the majority of classes containing at least 1 HIER are those at DIT level two. Between versions 4 and 5, the number of HIER remained constant. Between those versions, the total number of classes in the system remained the same.

The striking feature of both Figures 6.1 and 6.2 is the lack of HIER overall, despite significant additions of classes at certain points. Developers clearly avoided adding classes at lower levels of the hierarchy and favoured cross-hierarchy calls.

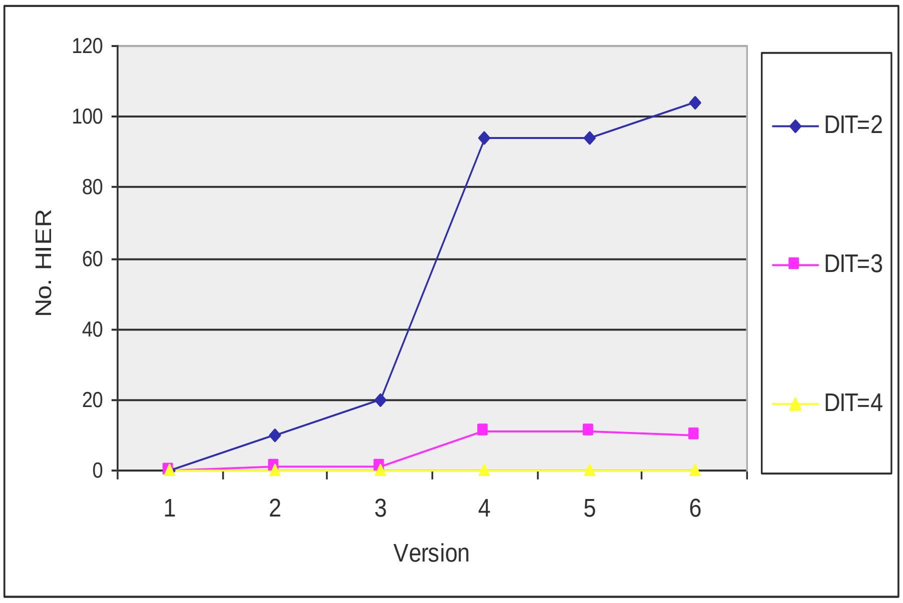


Figure 6.1. HIER frequencies HSQLDB (all versions)

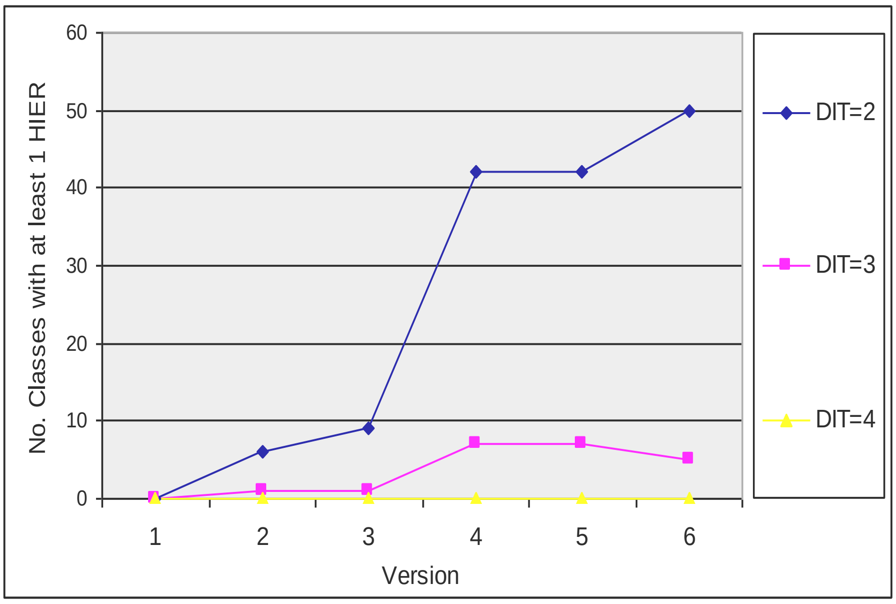


Figure 6.2. Number of classes with at least 1 HIER HSQLDB (all versions)

Figure 6.3 shows the net changes of HIER at each DIT level in the six versions of HSQLDB. A single peak can be seen in changes of HIER between versions 3 and 4. This is due to the sudden rise in number of classes in the system between versions 3 and 4. Interestingly, only 1 negative change was observed at DIT three between versions 5 and 6.

All other changes of HIER observed were positive. Class interaction within the hierarchy therefore tended to increase as the system evolved.

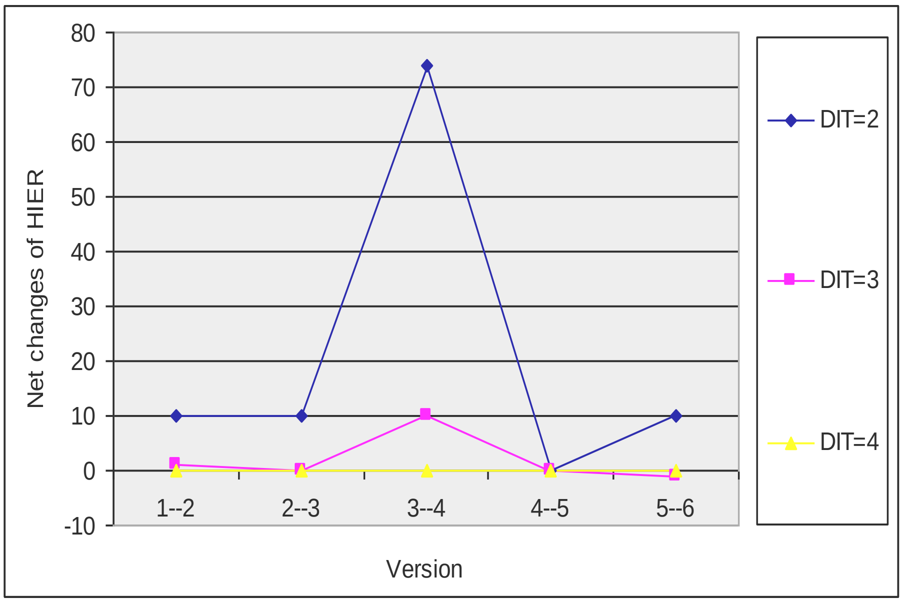


Figure 6.3. Net changes of HIER HSQLDB (all versions)

Figure 6.4 shows the frequencies of EXT and Figure 6.5 the number of classes with at least 1 EXT at each DIT level across six versions of HSQLDB. From Figures 6.4 and 6.5, we see that DIT one is where the majority of EXT exists. Again, the presence of large number of EXT may be influenced by the presence of large number of classes at DIT one. The contrast between the *x*-axis scales for Figure 6.1 and 6.2 and those of Figure 6.4 and 6.5 illustrate the differences between the values of HIER and EXT metrics, respectively and the tendency for developers to favour EXT rather than HIER.

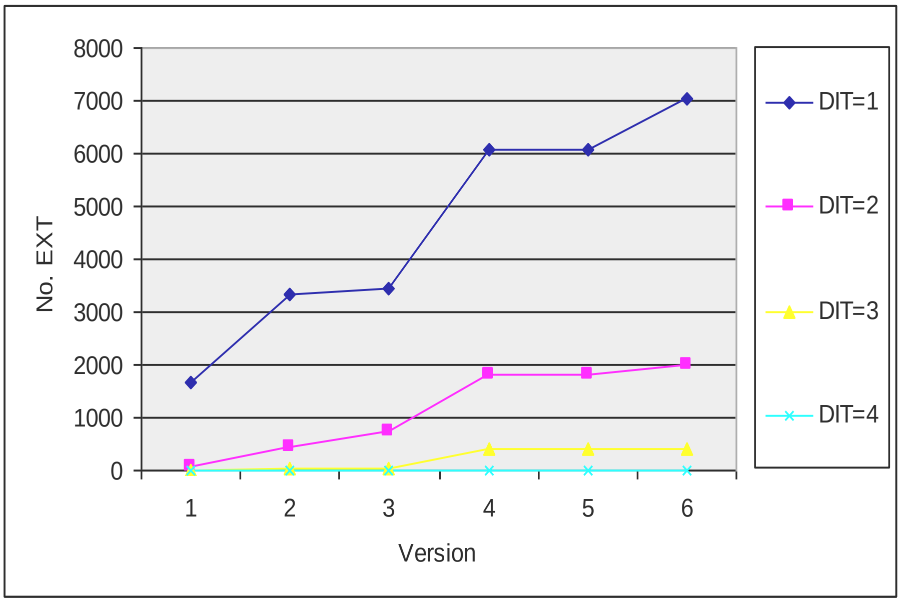


Figure 6.4. EXT frequencies HSQLDB (all versions)

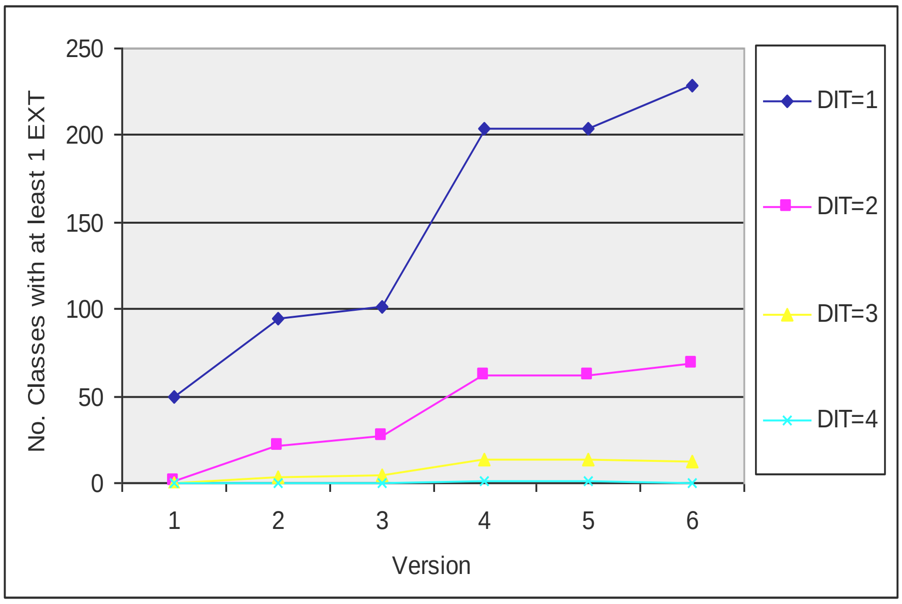


Figure 6.5. Number of classes with at least 1 EXT HSQLDB (all versions)

Figure 6.6 shows the net changes of EXT at each DIT level across the six versions of HSQLDB. Three peaks are visible: between versions 1 & 2, 3 & 4, and 5 & 6. The system seems to be relatively stable in the transition between both versions 2 & 3 and 4 & 5, suggesting that the system is changing in every second transition of versions. Only between versions 5 and 6 do we observe a negative growth in EXT (-5 from DIT three and -1 from DIT one) – all other changes were positive. The transition between versions 3 and 4 appears to be the point where significant changes are made to the system in terms of EXT values. (The total number of classes increased from 147 in version 3 to 323 in version 4.) The total HIER increased from just 21 to 105, while in the same period, the total number of EXT increased from 4229 to 8298.

In the transition from version 4 to 5 where the number of classes at DIT one decreased by 1, the total EXT at DIT one actually increased from 6081 to 6087, while the number of EXT at DIT two stayed constant. This demonstrates that even if the number of classes at a specific level falls, then this does not seem to influence the trend in EXT. It might well be that movement of classes around the hierarchy actually contributes to added EXT coupling, where previously that coupling was restricted to the subclass-superclass relationship. If we generally consider external calls to be potentially more dangerous, in terms of coupling and fault-proneness, then as the HSQLDB system evolved, relatively harmful forms of coupling became prominent.

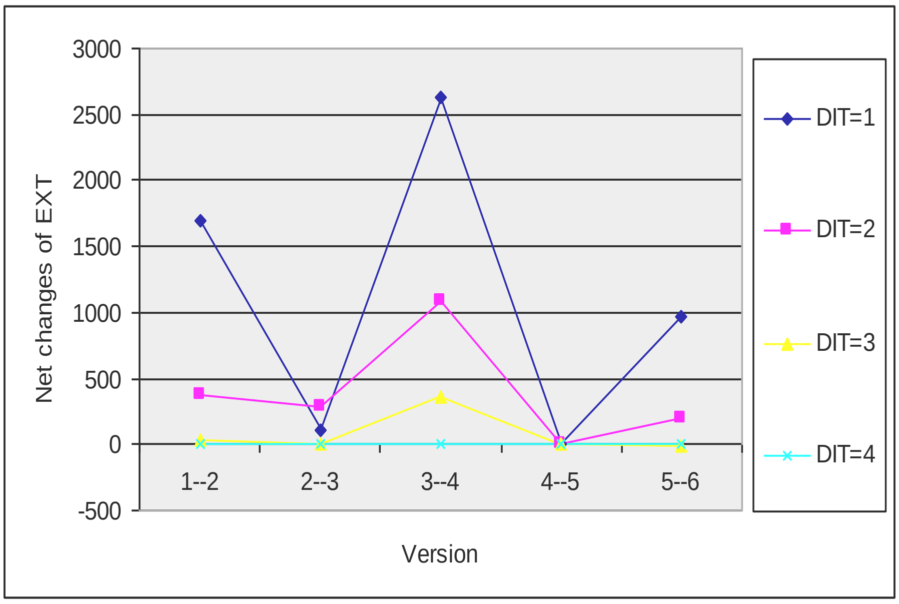


Figure 6.6. Net changes of EXT HSQLDB (all versions)

The maximum DIT found for the JasperReports system was 5 throughout the 12 versions of the system studied. Figure 6.7 shows the frequencies of HIER and Figure 6.8 the frequencies of the number of classes with at least 1 HIER across the 12 versions of the system. From Figure 6.7, all HIER values exist at DIT two and three. DIT four and five contained zero HIER throughout the lifetime of the system, implying that there was a lack of interaction between classes within the hierarchy at these low levels. This was a surprising result from the study, since we might have expected classes at deeper levels to be placed there to take advantage of superclass relationships. This does not seem to be the case, however. In (Nasseri et al. 2008), we found that JasperReport classes at DIT four and five showed zero changes in terms of number of classes throughout the set of 12 versions studied. Likewise, in (Nasseri and Counsell 2009b, Nasseri and Counsell 2009a) we observed that in the same system, classes at DIT four and five exhibited no change in terms of number of methods and attributes, respectively throughout the system's lifetime. Developers of this system undertook relatively little maintenance at deep levels of hierarchy. Perhaps, as previous studies suggest, a maximum of three levels of inheritance is more practical and manageable and more amenable to change, supporting Daly's view on use of inheritance at three levels (Daly et al. 1996). From Figure 6.7, a fluctuation in number of HIER at DIT two is visible, while the number of HIER at DIT three seems to remain relatively static.

From Figure 6.8, the number of classes with at least 1 HIER at each DIT level is also relatively static. The number of classes at DIT two with at least 1 HIER increased from 51 in version 1 to 58 in version 12 and the total number of HIER at DIT two increased from 156 in version 1 to 305 in version 12. This implies that, despite the number of classes with at least one HIER not increasing significantly, the number of HIER increased significantly; in other words, growth of HIER is not always influenced by the addition of classes; maintenance to existing classes seems to have just as large an effect.

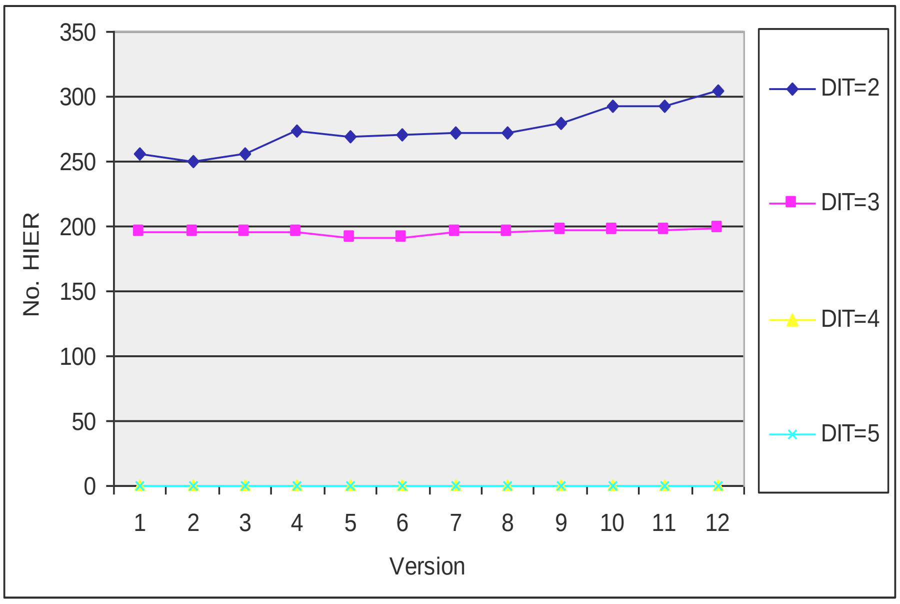


Figure 6.7. HIER frequencies JasperReports (all versions)

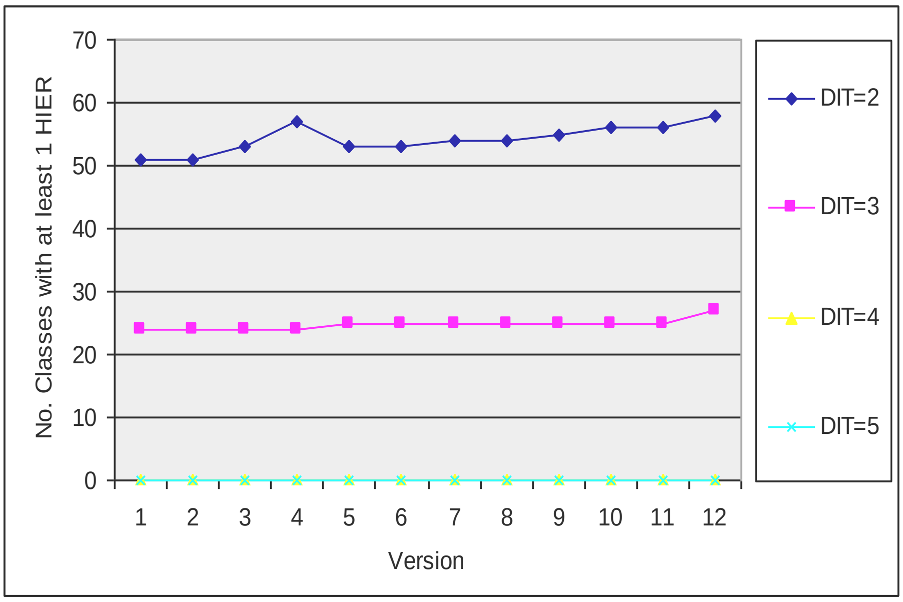


Figure 6.8. Number of classes with at least 1 HIER JasperReports (all versions)

Figure 6.9 shows the net changes of HIER at each DIT level in JasperReports. From Figure 6.9, three positive peaks in changes at DIT two are visible between versions 3 & 4, 9 & 10 and 11 & 12. In addition, we see a negative peak in changes of HIER at DIT two between versions 1 and 2. In JasperReports, the total HIER at DIT two and three across all versions was 3288 (i.e., 58.40%) and 2342 (i.e., 41.60%), respectively, implying that very little activity took place at levels below that. Classes at DIT four and five contained zero HIER. The evidence presented for JasperReports suggests that the majority of maintenance activity occurred at shallow levels of DIT.

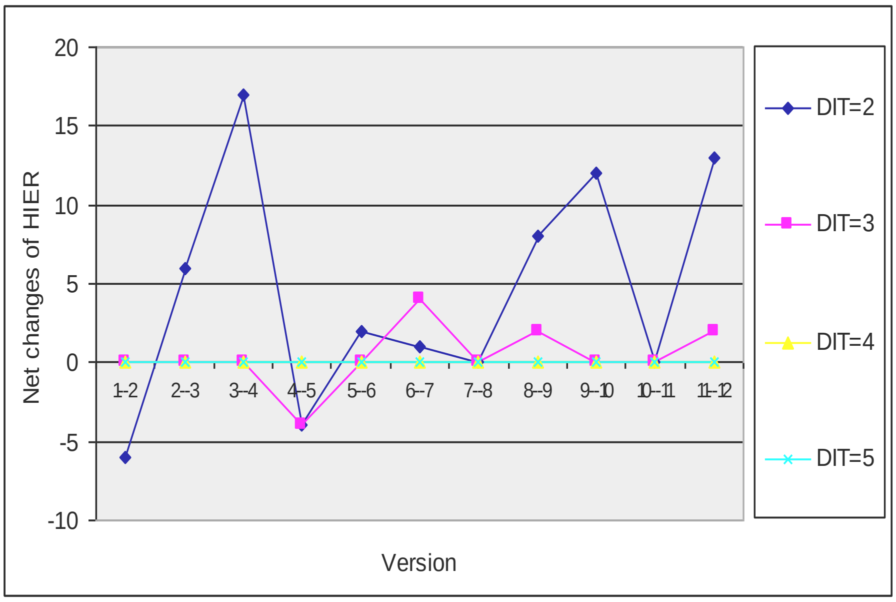


Figure 6.9. Net changes of HIER JasperReports (all versions)

Figure 6.10 shows the breakdown of number of EXT and Figure 6.11 the number of classes with at least 1 EXT at each DIT level in the 12 versions of JasperReports. From Figure 6.10, the majority of EXT tends to occur at DIT one and two. Overall, 174541 (i.e., 91.92%) EXT were at DIT one and two; 7829 (i.e., 91.98%) classes with at least 1 EXT were found at DIT levels one and two throughout the versions of the system. It is noticeable from Figures 6.10 and 6.11 that the number of EXT and the number of classes with at least 1 EXT rose after version 3. In versions 1, 2 and 3, the total number of classes with at least 1 EXT at DIT one remained at 380. However, the number of EXT in the same level changed significantly (from 7811 EXT in version 1 to 7902 EXT in version 2 and 7825 EXT in version 3) suggesting that class interaction may again change and be independent of net changes in number of classes (cf. Figure 6.9).

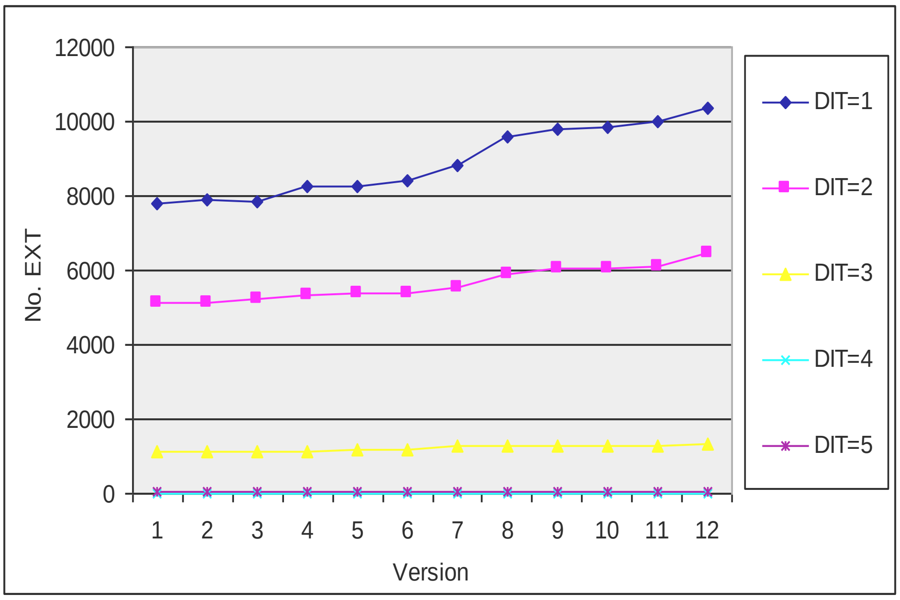


Figure 6.10. EXT frequencies JasperReports (all versions)

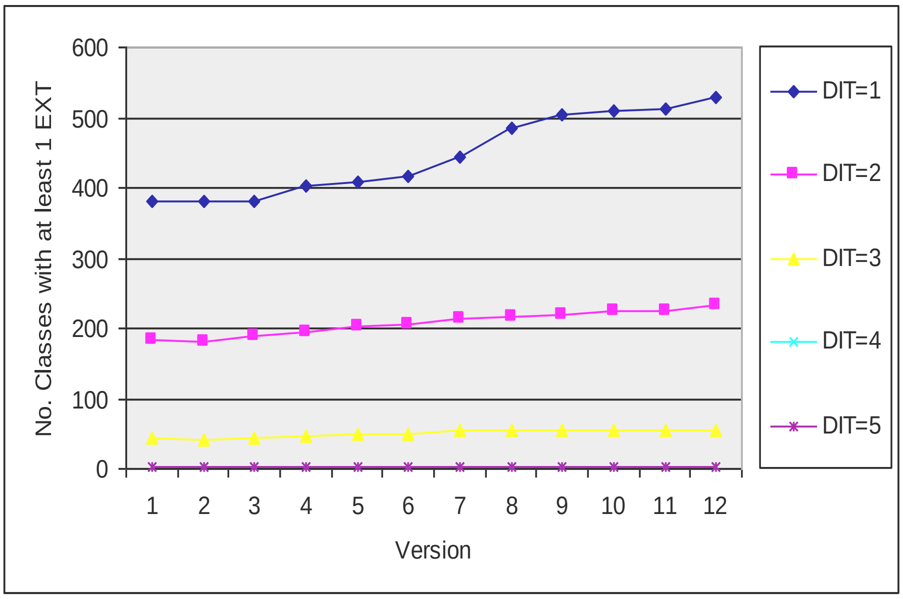


Figure 6.11. Number of classes with at least 1 EXT JasperReports (all versions)

Figure 6.12 shows the net changes (positive and/or negative growth) of EXT at each DIT level across the 12 versions of JasperReports. From Figure 6.12, three peaks in changes of EXT at DIT one are visible (between versions 3 & 4, 7 & 8 and 11 & 12). Overall 4071 (i.e., 95.25%) changes of EXT were made to classes at DIT one and two again suggesting that maintenance activity at level three and beyond was minimal. Moreover, the maximum change in EXT occurs between versions 7 and 8, despite the fact that the maximum change in number of classes occurs between versions 6 and 7.

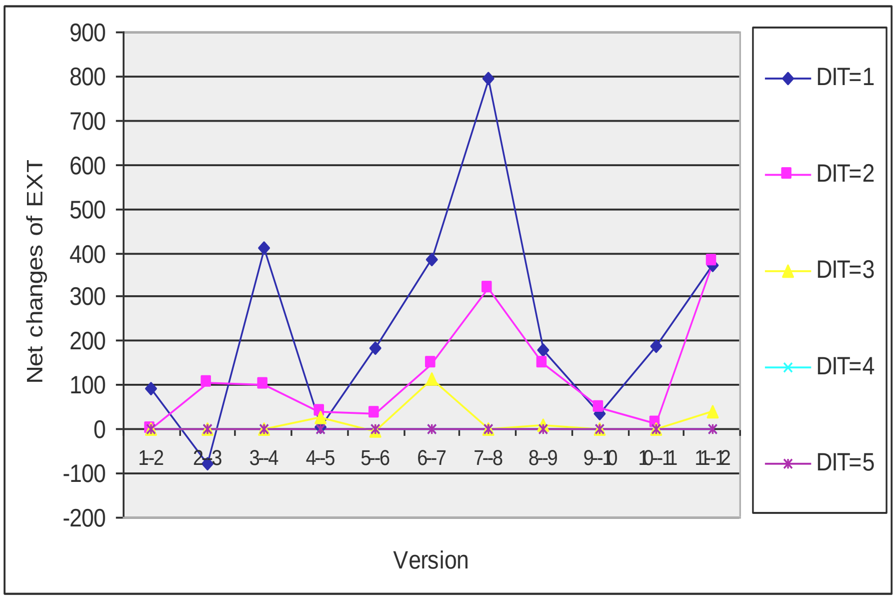


Figure 6.12. Net changes of EXT JasperReports (all versions)

The maximum DIT found for SwingWT was 7. Figure 6.13 shows the frequencies of HIER and Figure 6.14 the frequencies of the number of classes with at least 1 HIER in the set of 22 versions of SwingWT. From Figure 6.13, a fluctuation can be seen in number of HIER in every version. In version 1, we found only 9 HIER in the entire system, all of which were located at DIT three. DIT two contained zero HIER. It is also interesting from Figure 6.13 that in versions 21 and 22, DIT five is where the maximum HIER (112) exists. That sudden rise in number of HIER at DIT five is accompanied by a decrease in number of HIER at DIT three and four. This trend can also be seen in Figure 6.14 where in version 21 the number of classes with at least 1 HIER increases with a corresponding drop in number of classes with at least 1 HIER at DIT three and four. In terms of number of methods, we found that between versions 20 and 22 the number of methods at DIT five dropped from 719 to 690.

Between versions 20 and 22, the developers of the system therefore paid more attention to altering existing class interaction than simply adding new classes. In SwingWT, we found the majority of HIER at DIT two with 988 HIER (i.e., 28.87%) over 22 versions of the system. The number of HIER at DIT three, four, five, six and seven, across all versions of the system, was 450 (i.e., 13.15%), 670 (i.e., 19.58%), 644 (i.e., 18.83%), 306 (i.e., 8.94%) and 364 (i.e., 10.64%), respectively.

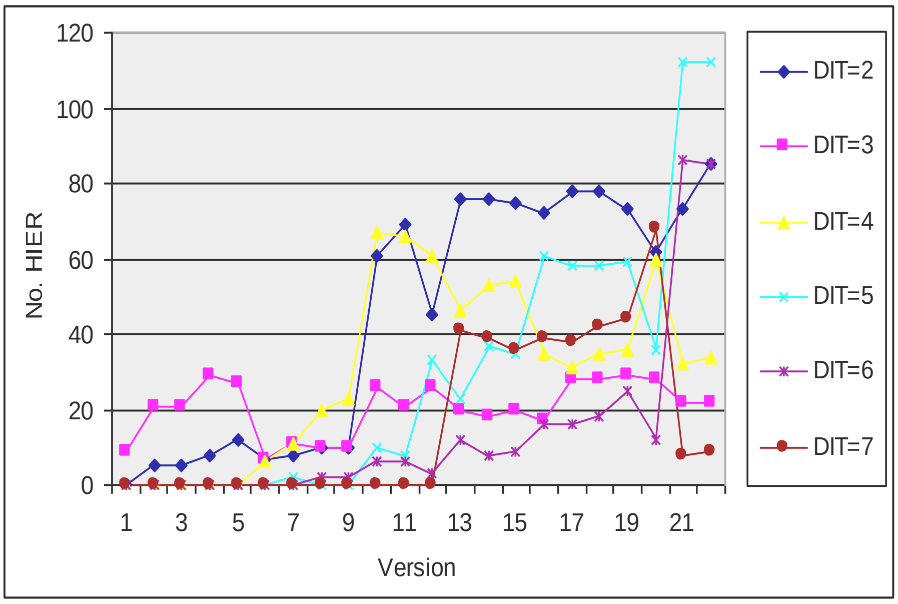


Figure 6.13. HIER frequencies SwingWT (all versions)

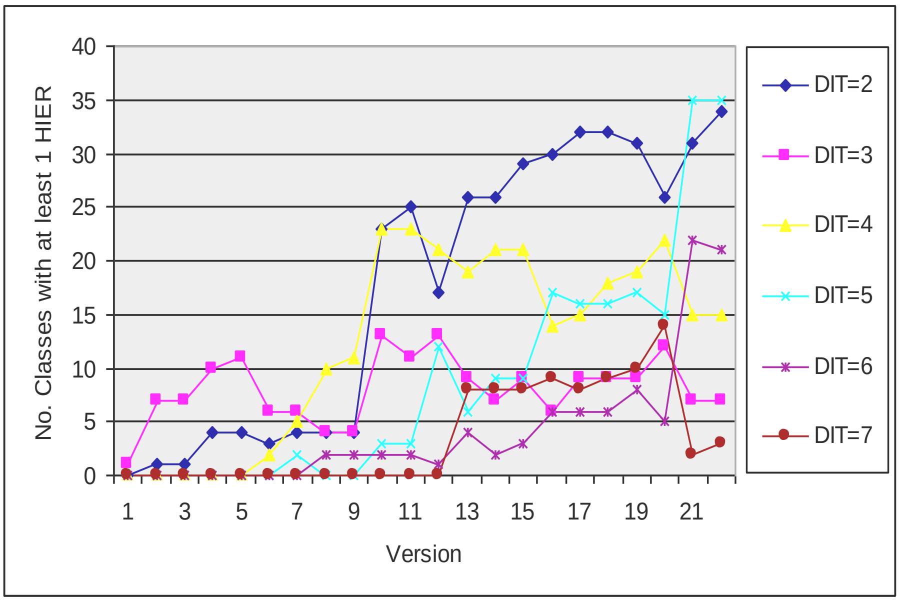


Figure 6.14. Number of classes with at least 1 HIER SwingWT (all versions)

Figure 6.15 shows the net changes of HIER at each DIT level in all versions of SwingWT. Between versions 20 and 21, there is a sudden increase in changes of HIER at DIT five and six with a corresponding decrease in changes of HIER at DIT seven. The SwingWT system showed some evidence of large numbers of classes being 'collapsed' to shallower levels which explains the sudden rise at one level and a decrease at another.

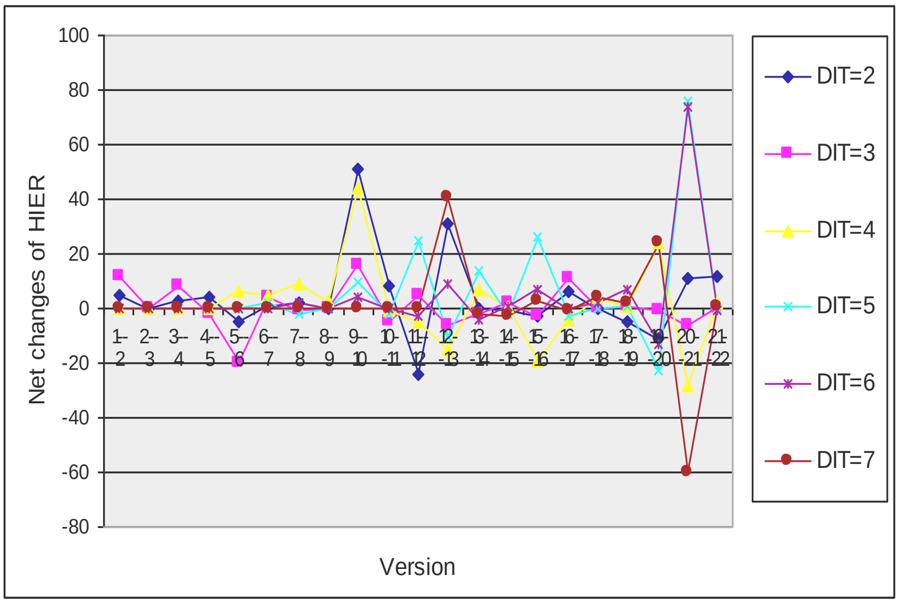


Figure 6.15. Net changes of HIER SwingWT (all versions)

Figure 6.16 shows the frequencies of number of EXT and Figure 6.17 shows the number of classes with at least 1 EXT in the entire set of 22 versions of SwingWT. From Figure 6.16, the majority of EXT exists at DIT one. In both figures, we see a sudden increase in EXT in version 10. In that version, the system grew in size by 160 (i.e., 69.57%) classes. The number of classes with at least 1 EXT in the same version increased by 107 (i.e., 99.07%) and the number of classes with at least 1 HIER increased by 43 (i.e., 204.76%). This latter analysis points to the possibility that the sudden rise in EXT and HIER in version 10 is due to the large number of new classes being added to the system.

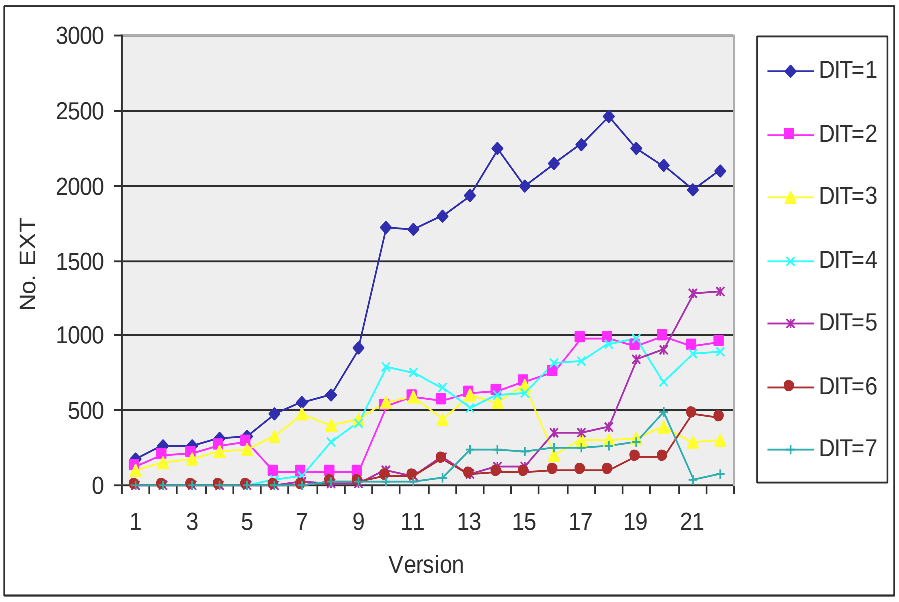


Figure 6.16. EXT frequencies SwingWT (all versions)

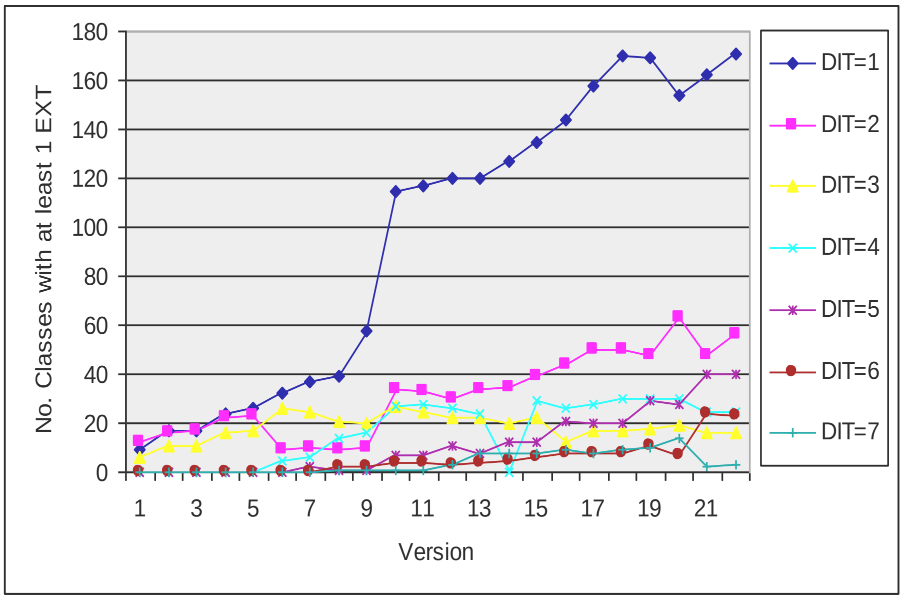


Figure 6.17. Number of classes with at least 1 EXT SwingWT (all versions)

Figure 6.18 shows the net changes of EXT in the 22 versions of SwingWT. We see a positive peak between versions 9 and 10. The transition between these versions was the point where the maximum number of classes (160) was added to the system. We also see a decrease in changes of EXT at DIT three between versions 15 and 16 with a corresponding rise in changes of EXT at DIT one, two, four and five. From Figure 6.18, it is also visible that between versions 20 and 21 the changes of EXT at DIT one, two, three and seven falls with a corresponding rise in changes of EXT at DIT four, five and six. From Figure 6.18, we also see that the changes of EXT at each DIT level are generally erratic, particularly at levels one, two, three and four.

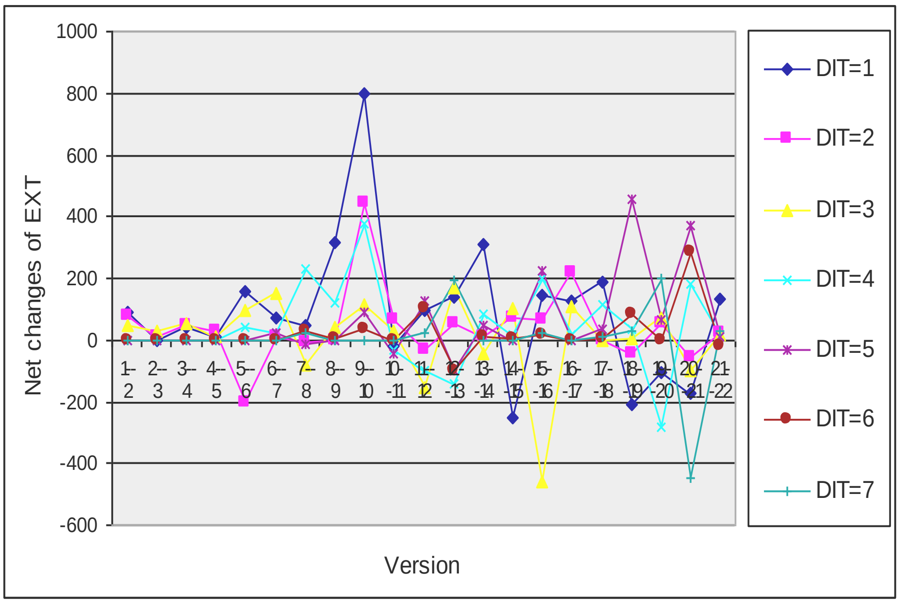


Figure 6.18. Net changes of EXT SwingWT (all versions)

The maximum DIT for Tyrant was 5 but only until version 4. In version 5, it fell to 3 and stayed constant throughout the 45 versions of the system. Figure 6.19 shows the frequencies of number of HIER at each DIT and Figure 6.20 shows the number of classes with at least 1 HIER at each DIT level in 45 versions of Tyrant. From these two figures, we see that the system underwent significant re-engineering between versions 4 and 5 - after which the system stabilized and showed no change for a significant number of versions. This trend was also found in previous studies by the same authors (Nasseri et al. 2008, Nasseri and Counsell 2009b, Nasseri and Counsell 2009a) where after version 5 Tyrant showed no change in terms of number of classes, methods and attributes, respectively.

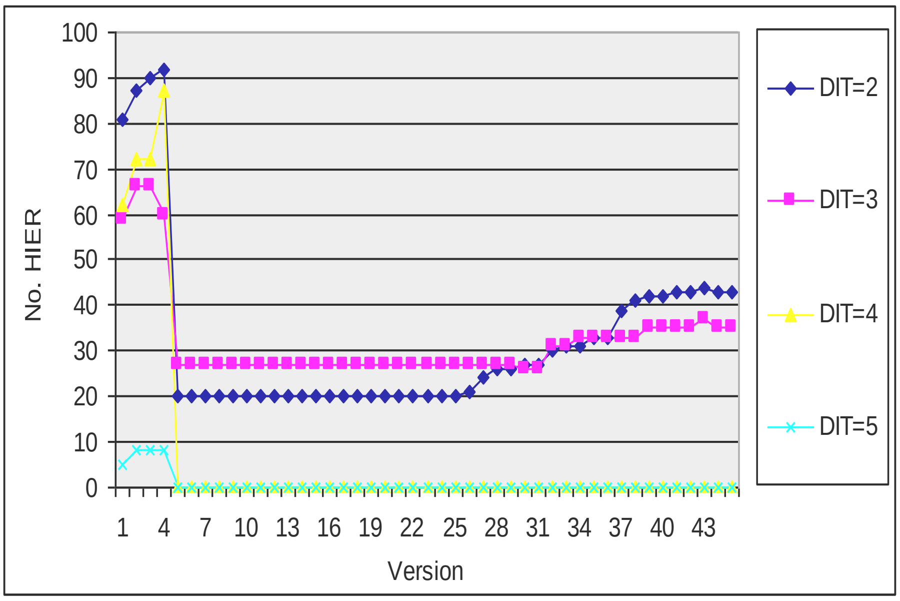


Figure 6.19. HIER frequencies Tyrant (all versions)

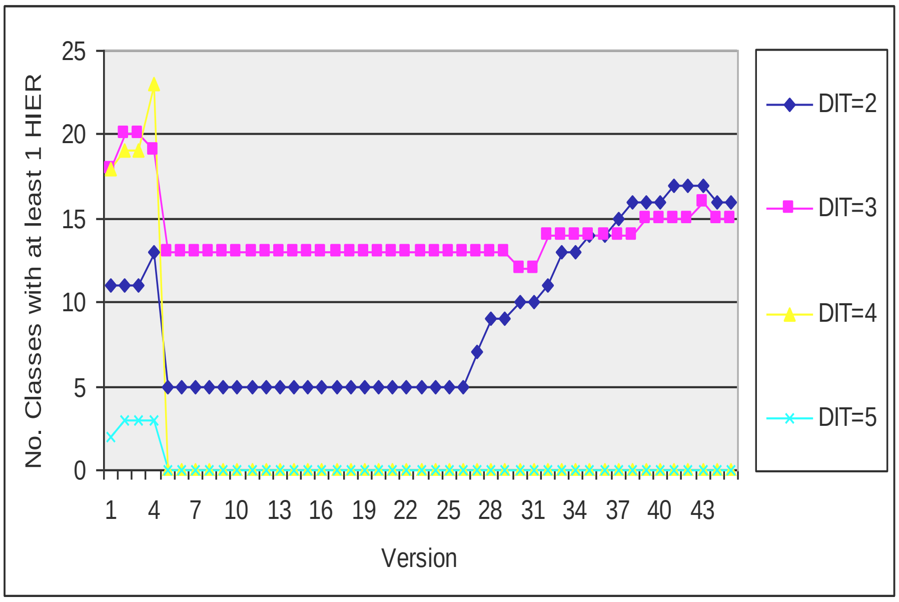


Figure 6.20. Number of classes with at least 1 HIER Tyrant (all versions)

Figure 6.21 shows the frequencies of net changes of HIER at each DIT level across 45 versions of Tyrant. We see a negative peak in number of HIER between versions 4 and 5 when the system underwent major re-engineering. After version 5, the changes of HIER seem to be negligible in comparison with the remaining three systems. Following version 5, the maximum change in number of HIER at both DIT two and three was only 8.

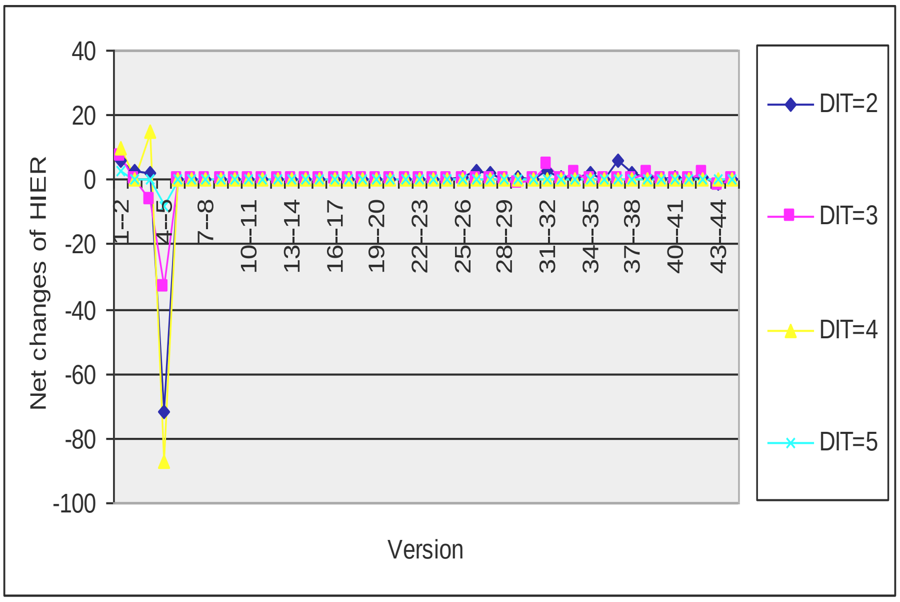


Figure 6.21. Net changes of HIER Tyrant (all versions)

Figure 6.22 shows the number of EXT at each DIT level and Figure 6.23 the number of classes with at least 1 EXT at each DIT level for the versions of Tyrant. From the two figures, the majority of EXT exists at DIT level one and, to a lesser extent, level three.

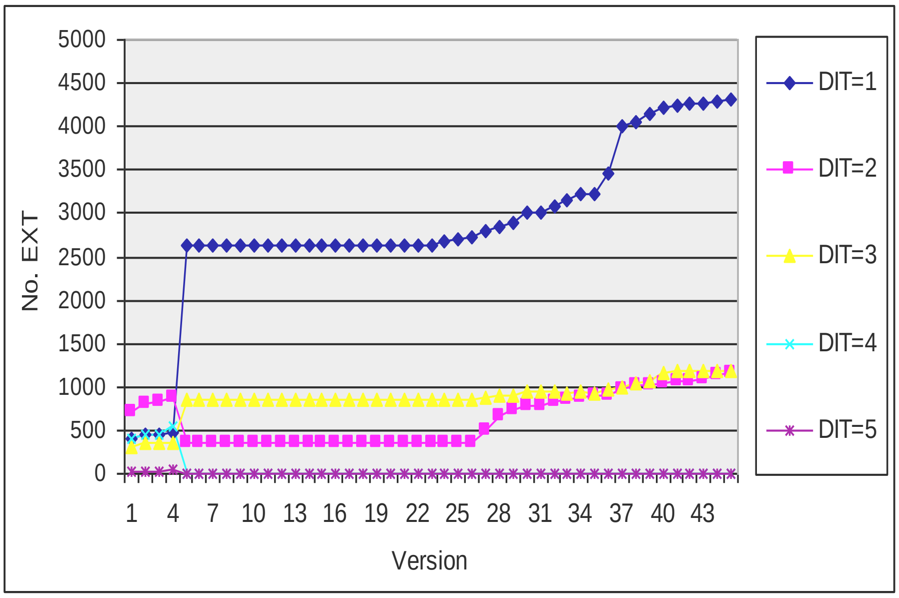


Figure 6.22. EXT frequencies SwingWT (all versions)

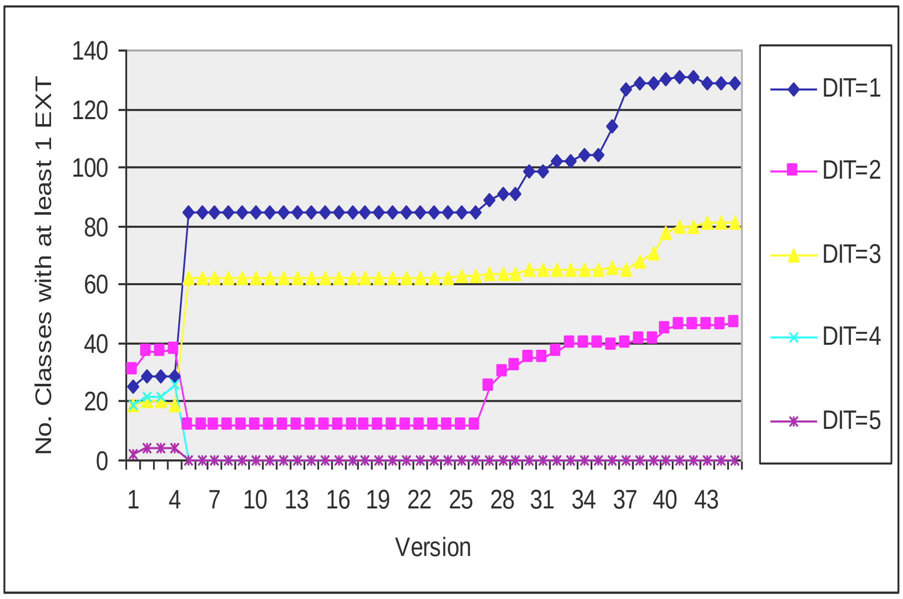


Figure 6.23. Number of classes with at least 1 EXT Tyrant (all versions)

Figure 6.24 shows the breakdown of net changes of EXT at each DIT level across 45 versions of Tyrant. From Figure 6.24, we see two positive peaks in changes of EXT at DIT one and three between versions 4 and 5 with two corresponding negative peaks in EXT changes at DIT two, four and five. Again, following major changes (between versions 4 and 5) the system stabilizes, in terms of EXT, for a significant number of versions.

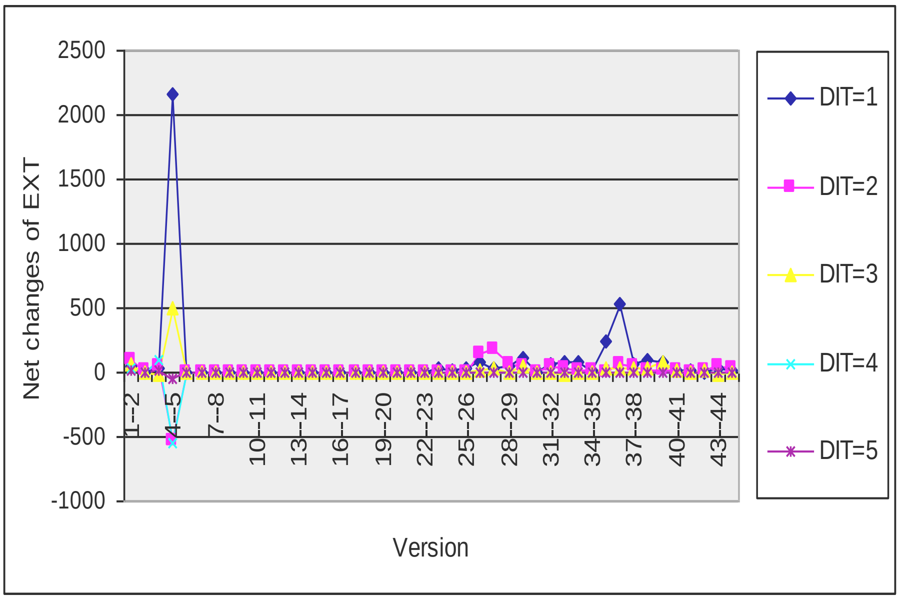


Figure 6.24. Net changes of EXT Tyrant (all versions)

6.5.2 Final version analysis

We now explore the specific features of each system in the latest version to establish how the values of EXT and HIER compare. We have chosen the latest version in each case, since differences between these two metrics are likely to be more pronounced as a system ages, and invariably decays. Table 6.2 shows the numerical values of HIER and EXT at each DIT level for HSQLDB. The table format (after column 1 for DIT) is as follows:

(1) frequency and percentage of classes (Classes),
(2) frequency and percentage of HIER (HIER),
(3) frequency and percentage of EXT (EXT),
(4) average number of HIER (HIER/Classes) and
(5) average number of EXT (EXT/Classes) at each DIT level.

From Table 6.2, the majority of HIER (i.e., 91.23%) exists at DIT level two. The average HIER at DIT two is also higher than that of DIT three. The number of EXT at DIT one is significantly higher than that of DIT two and three. The average EXT however, shows a different trend. The average EXT at DIT three seems to be significantly higher than that of average EXT at DIT one and two. This was a surprising result from our analysis. In theory, we would expect classes at higher levels of hierarchy (DIT one) to be more amenable to coupling, with other classes outside the hierarchy, than classes at lower levels. Classes at lower levels of hierarchy are more dependent on the functionality offered by their superclasses rather than classes outside the line of hierarchy.

DIT	Classes	HIER	EXT	HIER/Classes	EXT/Classes
1	279 (77.93%)	0	7050 (74.52%)	0	25.27
2	68 (18.99%)	104 (91.23%)	2012 (21.27%)	1.53	29.59
3	11 (3.07%)	10 (8.77%)	399 (4.22%)	0.92	36.27

Table 6.2. HIER and EXT at each DIT level in HSQLDB (final version)

Table 6.3 shows the same numerical values for the final version of JasperReports in the format of Table 2. From Table 6.3, the maximum number of HIER (i.e., 60.52%) exists at DIT two and to a lesser extent DIT three. However, on average the highest number of HIER (HIER/Classes) exists at DIT three. We also see that DIT four and five contain zero HIER. A principle of inheritance is that related functionalities should be encapsulated together and the classes providing those functionalities should interact with each other. The lack of HIER in classes at lower levels of hierarchy (DIT four and five) is in direct contradiction with this principle. In terms of EXT, the majority of EXT (i.e., 56.92%) tends to exist at DIT one. DIT four is where the minimum number of EXT exists. The average EXT however shows a different trend. On average, classes at DIT two and three, respectively, contain the highest EXT.

DIT	Classes	HIER	EXT	HIER/Classes	EXT/Classes
1	761 (69.31%)	0	10383 (56.92%)	0	13.64
2	258 (23.50%)	305 (60.52%)	6468 (35.46%)	1.18	25.07
3	65 (5.92%)	199 (39.48%)	1334 (7.31%)	3.06	20.52
4	10 (0.91%)	0	5 (0.003%)	0	0.5
5	4 (0.36%)	0	51 (0.28%)	0	12.75

Table 6.3. HIER and EXT at each DIT level in JasperReports (final version)

Table 6.4 shows the values of HIER and EXT for the final version of SwingWT. Despite the fact that the majority of classes reside at higher DIT levels, the maximum number of HIER is from classes at DIT five. It is notable that 28 classes (i.e., 4.52% of all classes) account for 112 (i.e., 32.28%) of HIER and 1289 (i.e., 21.27%) of EXT in the system. We see that the majority of EXT exists at DIT one; however, when taking into account the number of classes, DIT five, six and four, respectively, is where the majority of EXT exists. On average, classes at lower DIT levels have higher coupling, given by HIER and EXT, than classes at higher levels. While in theory we expected the majority of EXT to occur at higher levels (DIT one and two) of a hierarchy from a practical perspective, the evidence so far suggests that this is not always the case.

DIT	Classes	HIER	EXT	HIER/Classes	EXT/Classes
1	429 (69.19%)	0	2096 (34.59%)	0	4.89
2	99 (15.97%)	85 (24.50%)	956 (15.78%)	0.87	9.66
3	24 (3.87%)	22 (6.34%)	304 (5.02%)	0.92	12.67
4	25 (4.03%)	34 (9.80%)	890 (14.69%)	1.36	35.6
5	28 (4.52%)	112 (32.28%)	1289 (21.27%)	4	46.04
6	11 (1.77%)	85 (24.50%)	455 (7.51%)	7.73	41.36
7	4 (0.65%)	9 (2.59%)	70 (1.16%)	2.25	17.5

Table 6.4. HIER and EXT at each DIT level in SwingWT (final version)

Table 6.5 shows the numerical values of HIER and EXT for the final version of Tyrant. From Table 6.5, the majority of HIER (i.e., 55.13%) exists at DIT two. The average HIER

(HIER/Classes) also exhibits a similar trend. From Table 6.5, we see a strong tendency for average EXT (EXT/Classes) to exist at higher levels of DIT and the trend seems to be downwards as DIT increases. System design has serious implications on class interaction. The choice of method invocation in a class to the methods of another class is a design decision which may have a deteriorating impact on overall system functionality.

DIT	Classes	HIER	EXT	HIER/Classes	EXT/Classes
1	142 (50.01%)	0	4321 (64.79%)	0	30.43
2	49 (17.95%)	43 (55.13%)	1167 (17.50%)	0.88	23.82
3	82 (30.04%)	35 (44.87%)	1181 (17.71%)	0.43	14.40

Table 6.5. HIER and EXT at each DIT level in Tyrant (final version)

Table 6.6 shows how method calls are distributed in the final versions of the four systems. The column format of Table 6.6 (after column 1) is as follows:

1) number and the percentage of classes containing both HIER and EXT (HIER&EXT),
2) number and percentage of classes containing none (NONE),
3) number and percentage of classes containing only HIER (HIER) and
4) number and percentage of classes containing only EXT.

From Table 6.6, the majority of classes contain EXT. It was surprising that in the four systems we found zero classes containing only HIER. Remarkable from Table 6.6 is the large number of classes (286) in SwingWT containing zero HIER and EXT. SwingWT is a GUI application which consisted of a maximum of seven levels of DIT. We expected this system to be more amenable to method calls (both HIER and EXT). However, the evidence showed that this is not the case. Scrutiny of the data revealed that 6407 method calls, both HIER and EXT, were spread across 334 classes. On average, each class had 19.18 method calls.

In comparison to the remaining three systems (HSQLDB: 30.92, JasperReports: 22.75 and Tyrant 26.25) SwingWT contained the minimum number of method calls per class. From a coupling perspective this may sound encouraging, but from a design perspective method calls are not evenly spread to all parts of the system, with the 46.13% of classes containing zero method calls (both HIER and EXT). This latter result confirms previous findings (Counsell et al. 2006a, Advani et al. 2006) that Swing has been poorly built and consistently shows features indicating that the system is decaying and has been 'patched up'.

Systems	HIER&EXT	NONE	HIER	EXT
HSQLDB	55 (15.36%)	48 (13.41%)	0 (0%)	255 (71.23%)
JasperReports	85(7.74%)	274(24.95%)	0(0%)	739(67.30%)
SwingWT	115(18.55%)	286(46.13%)	0(0%)	219(35.32%)
Tyrant	31(11.36%)	16(5.86%)	0(0%)	226(82.78%)

Table 6.6. The HIER and EXT data in the four systems (final versions)

6.5.3 Mann-Whitney U-test and Correlation Analysis

6.5.3.1 Class Cohesion Analysis

We now investigate the impact of method calls (both HIER and EXT) on class cohesion, given by the Lack of Cohesion metric of Chidamber and Kemerer (Chidamber and Kemerer 1994) in the final versions of the four studied systems. A high value of LCOM in a class suggests high complexity which potentially enhances class vulnerability to faults. To investigate the impact of method calls (both HIER and EXT) on class cohesion, we carried out two, one-tailed Mann-Whitney U-tests (Hinkle et al. 1995). We developed and tested the following hypotheses in order to explore the impact of method calls on class cohesion.

Null hypothesis HO1: The cohesion of a class is not influenced by HIER in that class
Alternative hypothesis HA1: HIER in a class tends to decrease the cohesion of that class.

Null hypothesis H02: The cohesion of a class is not influenced by EXT in that class.
Alternative hypothesis HA2: EXT in a class tends to decrease the cohesion of that class.

Table 6.7 shows the results of the Mann-Whitney U-test carried out on HIER and LCOM for the final versions of the four systems. The format of the table is as follows: 1) the two samples or groups of the classes (i.e., classes containing HIER (HIER) and classes without HIER (Un-HIER)), 2) N - the number of classes in each group, 3) the mean-rank of scores within each group (Mean-Rank), 4) the total sum of rank within each group (Sum of Rank), 5) test statistics for the test (Mann-Whitney U-test), 6) the probability of test significance (p-Value) and 7) Z-Score: used to assess the significance of the test when either/both samples > 20 (i.e., if the Z-Score is > 1.96, irrespective of its sign, then the test is significant at the one/five percent level based on the p-Value).

Samples	N	Mean-Rank	Sum of Rank	M-Whitney U	p-Value	Z-Score
HIER	286	1304.40	373057	257716	0.000	-3.566
Un-HIER	2062	1156.48	2384669	◆	◆	◆

Table 6.7. Results of the Mann-Whitney U-test of HIER and LCOM

From Table 6.7, the mean rank value for classes with HIER (HIER row in Table 6.7) is higher than its respective mean rank value for classes without HIER (Un-HIER row in Table 6.7) and the Z-Score (-3.566) suggests that the test is significant at the 1% level (i.e., p-Value < 0.01). The results in Table 6.7 indicate that classes with HIER tend to have higher LCOM than classes without HIER. Table 6.8 shows the results of the Mann-Whitney U-test carried out on EXT and LCOM for the final versions of the four systems in the same format as Table 6.7.

Samples	N	Mean-Rank	Sum of Rank	M-Whitney U	p-Value	Z-Score
EXT	1728	1263.18	2182776.50	382439.5	0.000	-10.912
Un-EXT	620	927.34	574949.50	◆	◆	◆

Table 6.8. Results of the Mann-Whitney U-test of EXT and LCOM

From Table 6.8, the mean rank value for classes with EXT (EXT row in Table 6.8), is higher than its corresponding mean rank value for the classes without EXT (Un-EXT row in Table 6.8). The Z-Score (-10.912) suggests that the test is significant at the 1% level (i.e., p-Value < 0.01). The results in Table 6.8 indicate that classes with EXT tend to have higher LCOM than classes without EXT.

Based first on the evidence from Table 6.7, the Z-Score and p-Value confirm that the results are statistically significant. We are therefore in the position to refute H01 in favour of HA1 that HIER in a class tends to decrease the cohesion of that class. The evidence presented in Table 6.8 suggests that classes with EXT have higher LCOM than classes without EXT; we hence reject the H02 in favour of HA2; that EXT in a class tends to decrease the cohesion of that class.

6.5.3.2 Class Size Analysis

We now speculate that larger classes, given by the number of methods (NOM), tend to have higher HIER and EXT. To investigate this, we conducted a 2-tailed parametric (Pearson's) and two 2-tailed non-parametric (Kendall's and Spearman's) cross correlations of NOM versus HIER and EXT in final versions of the four systems. Table 6.9 shows the correlation values for the four systems.

Systems	Pearson's	Kendall's	Spearmen's
NOM vs. HIER	0.240**	0.195**	0.237**
NOM vs. EXT	0.615**	0.398**	0.525**

Table 6.9. Correlation of NOM versus HIER and EXT

From Table 6.9, all values are double asterisked indicating that the correlations are significant at the 1% level. The size of a class, given by NOM, is strongly correlated with method calls (both HIER and EXT). Given the evidence in Table 6.9, we can assert that the size of a class is strongly correlated with the *fan-out* (given by HIER and EXT) of a class. Classes with more methods contain more functionality and hence are more amenable to interaction with other classes in the system.

6.6 Discussion

We begin with the threats to the validity of the study and how we defend those threats. We start with the construct validity (i.e., the degree to which the measured concept can be measured accurately in a different way): method invocation is an essential element of OO systems which introduces coupling and should be measured accurately. In defence of this threat, we argue that the metrics used in this chapter measure method invocation from two distinct perspectives by distinguishing between method invocations within the hierarchy (HIER) and External method calls (EXT). The rationale for using HIER was to investigate method invocation from an inheritance perspective. That is, to explore whether, in practice, classes take advantage of the functionality offered by their superclasses and how class interaction within an inheritance hierarchy evolves as the system evolves. We found no other metric to measure method calling within inheritance hierarchy. The rationale behind using EXT was to measure method calling in classes outside the class hierarchy, and make a comparison to that of HIER. We appreciate that there are other metrics that measure coupling (which includes method calling) however, we required a metric to measure method invocation and exclude HIER and other forms of coupling. We therefore believe HIER and EXT metrics both capture class interaction in OO systems in a logical and structured fashion. Using HIER and EXT to measure method calls sufficiently supports the construct validity of the study. In terms of *external* validity (i.e., the degree to which the results of the study can be generalized): the set of systems used are OSS rather than proprietary systems. In our defence, we argue that the set of four systems analyzed are from various application domains ranging from a database system to a game engine and a GUI framework with various sizes and number of versions. Furthermore, we focused our analysis on OSS which has been the subject of many empirical studies (Capiluppi et al. 2004, Capiluppi and Ramil 2004, Counsell et al. 2006a, Advani et al. 2006, Nasseri et al. 2008, Nasseri and Counsell 2009a, Nasseri and Counsell 2009b). We believe these points sufficiently support the external validity of our study.

6.7 Summary

In this chapter we have presented an empirical analysis of evolution of four Java OSS from a method invocation perspective. We distinguished between method invocation within the hierarchy (HIER) and external method calls (EXT) in classes of the studied systems. The JHawk tool was used to extract the HIER and EXT from multiple versions and NOM and LCOM from final versions of the four systems. The evidence suggests that the majority of HIER and EXT respectively existed at DIT two and one. However, when considering the number of classes at each DIT level, no clear pattern could be observed as to where the majority of HIER and EXT existed. Similarly, we found that higher DIT levels (DIT one and two) tended to have a higher growth rate in HIER and EXT.

The evidence indicates that the majority of method invocation (both HIER and EXT) are made to the methods of the classes where the majority of functionality exists, irrespective of the position of classes within the hierarchy. The results also suggest that method invocation (both HIER and EXT) tends to detract from class cohesion and that class size, given by number of methods, is positively correlated with class coupling, given by (HIER and EXT). The results may be of interest to software developers/practitioners as to how classes in a system interact and how that interaction changes as a system evolves.

CHAPTER 7 “Warnings” and potential refactorings

7.1 Introduction

In the previous chapter we demonstrated how classes in an inheritance hierarchy interact with each other and how that interaction evolves as opposed to that of system evolution. In this chapter, we use data extracted by an automated tool called FindBugs (FindBugs 2008) to explore the potential “warnings/problems” embedded in systems as they evolve. To assist our analysis, we also collected an inheritance-based metric using the JHawk tool (JHawk 2008). We analyzed the frequency and type of warnings across both single and multiple versions of three Java OSS. The analysis allowed us to compare the types of warnings common to classes added at level one with those at other levels. It also allowed us to investigate the potential for refactoring elements which, in future versions may be problematic from a fault perspective. Our research investigates facets of the Java inheritance hierarchy that may store up problems as a system evolves.

In Section 7.2 we present the motivation for our empirical study. We then describe the study design including the OSS used, independent and dependent variables and research questions (Section 7.3). In Section 7.4 we present data analysis based on DIT, NOC, warnings and refactoring, before providing a brief discussion of the empirical study (Section 7.5); finally in Section 7.6 we provide a summary of our empirical results.

We note that part of the research in this chapter has been published in (Nasseri and Counsell 2008).

7.2 Study Motivation

Clearly, there is conflicting evidence about the use of inheritance and the benefits it may or may not bring in the past literature. However, the main thrust of evidence seems to be to avoid deep levels of inheritance whenever possible. In this chapter, we empirically investigated inheritance and warnings that may potentially create faults in a system and possible remedies through techniques such as refactoring. The main motivation for the study arose from the need to confirm/refute the claims previously made by many studies of inheritance and its effect on class fault-proneness.

7.3 Study Design

7.3.1 The three open-source systems

The three OSS used for the research in this chapter are JColibri, JasperReports and SwingWT (see systems 2, 3 and 5 in Section 2.3.2.5). The remaining 5 OSS in our system archive and a range of forty other Java OSS currently available from sourceforge.net were selected and investigated on a stratified sample basis, but very few exhibited significant numbers of warnings (problems embedded in a system that may lead to faults).

7.3.2 Independent and dependent variables

Warnings were collected using the FindBugs tool (FindBugs 2008). FindBugs uses static analysis of source code to extract the following six main categories of warnings.

1. Bad Practice (BP): The code does not follow endorsed coding practice (e.g., confusing method name, bad cast of object references and database resource not closed on all paths).

2. Correctness (CO): An unexpected mistake is found in a piece of code (e.g. null pointer de-references, suspicious calls to generic container methods and dropped exception).

3. Malicious Code Vulnerability (MCV): A situation where internal information is exposed or changed (e.g., method returning array may expose internal representation, mutable static field and storing reference to mutable object).

4. Multithreaded Correctness (MTC): Inappropriate use of thread (e.g., unsynchronized get method, synchronized set method and field not guarded against concurrent access).

5. Performance (PF): Warnings that aggravate system performance (e.g., unread field, wrong map iterator and private method never called).

6. Dodgy (DG): Code written in a confusing way that may cause faults (e.g., dead store to local variable, runtime exception captured and switch case falls through).

The warning data extracted by FindBugs was used as the dependent variable in this study. The inheritance measures 'Depth of Inheritance Tree' (DIT) and 'Number Of Children' (NOC) (see metrics 1 and 4 in Section 2.3.2.6) were collected using the JHawk tool (JHawk 2008) (described in Section 2.3.2.6) and were used as the independent variables in our study against which the propensity for warnings could be measured.

We accept that warnings may have different characteristics to manifestation of 'real' faults obtained through running and testing the code and used in some other studies (Arisholm and Briand 2006, Ostrand et al. 2004) and we also accept that false positives and false negatives are key threats to the validity of the study. A false positive/negative refers to the possible errors that may be made during an investigation. In the context of FindBugs a false positive refers to a situation when a warning is discovered by FindBugs which in fact is not a problem and may not lead to a fault. A false negative on the other hand refers to a situation when there may be a problem that may lead to a fault but FindBugs may not consider it as a problem. However, we believe that there is real value in understanding areas of code that 'could' cause maintenance problems at a later date. The areas of code identified by the FindBugs tool can also highlight prime sources of 'preventative' refactoring effort (Fowler 1999) associated specifically with inheritance. In addition, we also appreciate that detailed information on faults (i.e., which class a particular fault emerges from and which version a set of faults belong to) for OSS is also not available.

A previous study by Counsell et al. (Counsell et al. 2006a) has shown that inheritance-related refactorings (e.g., ‘Extract Subclass’ and ‘Extract Superclass’) are rarely undertaken in Java systems and the complexity associated with those refactorings may be a contributing factor. If sufficient warning signals of ‘code smells’ (Fowler 1999) can be provided early on in systems evolution, then FindBugs may help to solve an ongoing question for developers: ‘what do we refactor and when?

7.3.3 Research questions

We investigated the following research questions to explore the relationship between inheritance and warnings in the three systems.

1) Does the position of a class in inheritance hierarchy, according to its DIT, influence the number of warnings generated as a system evolves? If we can demonstrate that classes at deep levels of inheritance generate more warnings (on average) than those at relatively shallow levels, then that would support the view that developers should avoid deep levels.

2) Does the number of subclasses of a class, according to its NOC, influence the number of warnings generated in that class? In theory, we would expect a class with many children (subclasses) to have been tested more rigorously than classes with a lower NOC and hence to generate potentially fewer warnings. The extent of reuse of a class associated with many children also supports the argument for fewer warnings. However, we suggest that large-scale avoidance of deep inheritance levels in a system will cause potential faults to be found in classes with relatively low NOC values.

3) Does the type of warnings, extracted by FindBugs, inform any refactoring effort that could be applied to the classes as they evolve? We suggest that the warnings at DIT level one may give rise to a different set of potential refactorings than those at deeper levels.

7.4 Data analysis

7.4.1 DIT and warning analysis

Table 7.1 shows maximum (Max.), median (Med.), mean (Mn.) and standard deviation (STDEV) DIT values for the *latest versions* of the three systems; it also shows the total number of classes (Class) and total number of generated warnings (Warn.) for those systems.

Systems	Max.	Med.	Mn.	STDEV	Class	Warn.
SwingWT	7	1	1.67	1.28	620	368
JasperReports	5	1	1.40	0.68	1098	301
JColibri	2	1	1.04	0.18	228	74

Table 7.1. Summary DIT data for the three systems (final versions)

From Table 7.1, the maximum DIT belongs to SwingWT (7) suggesting extensive use of inheritance in this system. SwingWT is a GUI application and there is evidence from previous studies to suggest that, in contrast to many other types of system, GUI applications use inheritance extensively (Bieman and Zhao 1995). Similarly, SwingWT has the highest number of warnings despite the fact that it is not the largest system in terms of classes (it has over half the number of those in JasperReports). There is also empirical evidence to suggest that the Swing system has been poorly maintained, contravenes many OO practices and, consequently, has deteriorated badly over the course of the versions investigated (Counsell et al. 2006a, Advani et al. 2006) - this seems to be reflected in the number of warnings for this system.

Table 7.2 shows the numerical values for warnings found at each DIT level for the SwingWT system. The column format of Table 7.2 (after column 1 for DIT) is as follows: 1) the number of classes at each level of inheritance (Classes), 2) the number of warnings at each level (Warnings), 3) the average number of warnings per class at each level of inheritance (Warnings/classes), 4) the percentage of classes at each level (Class%), 5) the percentage of warnings at each level (Warning%).

From Table 7.2, approximately 59% of all warnings for SwingWT were found at DIT level one. Only 9.78% of warnings arose from classes at DIT two, and 2.72%, 7.88%, 15.76%, 2.72% and 1.36% warnings from DIT three, four, five, six and seven, respectively. However, in terms of the average number of warnings per class, Table 7.2 shows a significant rise in the numbers per class between levels three and five, after which there is a significant fall. We could suggest that for the SwingWT system, there may well be a 'good and bad' DIT range with threshold values at each end - warnings rise rapidly after DIT three, before falling after reaching DIT five. This theory is supported by the research of Daly et al. (Daly et al. 1996) where three levels of inheritance were found easier to modify than systems with no inheritance. Systems with five levels of inheritance, however, were shown to take longer to modify than the systems without inheritance.

DIT	**Cl.**	**Warn.**	**Warn./Cl.**	**Cl.%**	**Warn.%**
1	429	220	0.51	69.19	59.78
2	99	36	0.36	15.97	9.78
3	24	10	0.42	3.87	2.72
4	25	29	1.16	4.03	7.88
5	28	58	2.07	4.52	15.76
6	11	10	0.91	1.77	2.72
7	4	5	1.25	0.65	1.36

Table 7.2. Warnings/DIT (SwingWT)

Figure 7.1 shows the frequency of warnings for the JasperReports system; 257 (i.e., 85.38%) of warnings were found in classes located at DIT one. Only 38 (i.e., 12.62%) and 6 (i.e., 1.99%) of warnings were found in classes located at DIT two and three, respectively. Classes located at levels four and five exhibited no warnings whatsoever, but this might have been expected with the relatively small number of classes at these levels (i.e., 1.27% in total).

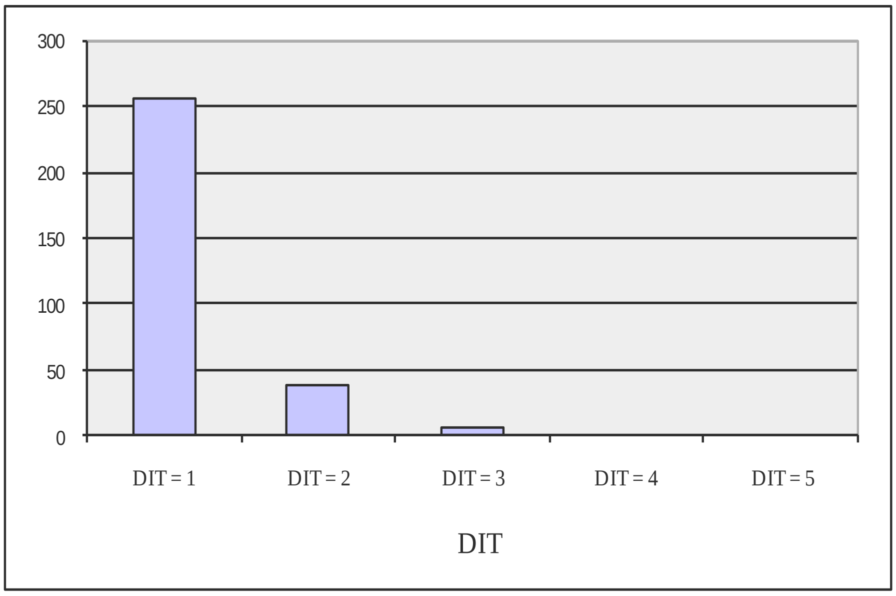


Figure 7.1. Warnings/DIT (JasperReports)

Table 7.3 presents the numerical data for JasperReports system (in the same format as for SwingWT). The data suggests that classes residing deeper in an inheritance hierarchy tend to be less potentially prone to warnings than classes located higher up in that hierarchy.

DIT	Cl.	Warn.	Warn./Cl.	Cl.%	Warn.%
1	761	257	0.34	69.31	85.38
2	258	38	0.15	23.50	12.62
3	65	6	0.09	5.92	1.99
4	10	0	0	0.91	0
5	4	0	0	0.36	0

Table 7.3. Warnings/DIT (JasperReports)

From Table 7.3, there are high numbers of classes at DIT two (i.e., 23.50% of all classes), yet these classes account for only approximately 12% of warnings. Over 85% of warnings occurred at DIT one. For this system, the larger the DIT is, the fewer the number of warnings there are. Only approximately 6% of classes reside at DIT three, and these account for only 1.99% of all warnings for this system. Clearly, for the JasperReports system and, to a lesser extent, the SwingWT system, the data suggests that the majority of warnings tend to reside where the majority of functionality is invested and this tends to be at DIT one. Table 7.4 shows the profile for the JColibri system in the same format as that of Tables 7.2 and 7.3. It shows that 74 (i.e., 100%) of warnings were found in classes located at DIT one (classes with DIT two exhibited zero warnings).

DIT	Cl.	Warn.	Warn./Classes	Class%	Warn.%
1	220	74	0.34	96.49	100
2	8	0	0	3.50	0

Table 7.4. Warnings/DIT (JColibri)

7.4.2 Distribution of warnings

One issue that arises as a result of our analysis is the distribution of warnings across classes. In other words, is the total set of warnings identified by the tool spread across a relatively few or the majority of the classes at the different levels for each of the three systems?

Table 7.5 shows the number of classes at each level with at least one warning and the percentage of the total number of classes in the system that this represents. For example, 102 classes at DIT one in the SwingWT contained at least one warning (i.e., 16.45% of all SwingWT classes). This also means that, since there are 429 classes in total at DIT one, approximately 25% of all classes at DIT one for this system contained a warning. The same table shows a clear bias towards warnings being generated at DIT one, for each of the three systems. Over 11% of all classes in JasperReports (125) contained at least one warning, significantly more than the total for all DIT levels two to five (the same applies to the JColibri system, where all the warnings were found to reside at DIT one).

Levels	SwingWT	JasperReports	JColibri
DIT1	102(16.45%)	125 (11.38%)	46(20.18%)
DIT2	22 (3.55%)	10 (0.91%)	0 (0%)
DIT3	7 (1.13%)	4 (0.36%)	-
DIT4	12 (1.93%)	0 (0%)	-
DIT5	11 (1.77%)	0 (0%)	-
DIT6	6 (0.97%)	-	-
DIT7	2 (0.32%)	-	-

Table 7.5. Classes/warnings at DIT levels

7.4.3 DIT and warning evolution analysis

Our analysis until now has looked at features of the three systems using the latest version as a basis. One feature of the three systems that emphasizes the role that DIT and associated warnings is how and where classes have been added over the course of the versions studied.

Figure 7.2 shows the changes in DIT (and the number of classes added over the course of the 22 versions) for SwingWT. The figure was also presented in Chapter 3 (Figure 3.4). It shows a steady rise in classes at DIT one and two from a relatively low level in version 1 to a high level in version 22. The same activity is not present at other levels, which remain relatively static.

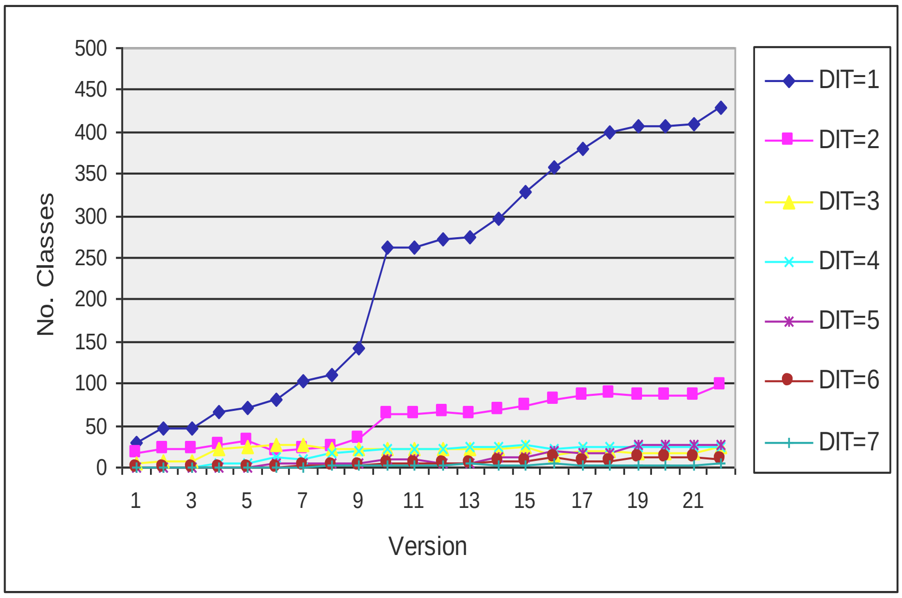


Figure 7.2. DIT frequencies SwingWT (all versions)

Figure 7.3 shows the number of classes and generated warnings for the SwingWT system throughout the versions studied. No relationship is clear from the graph, suggesting that new classes do not tend to attract any of the warnings; it may simply be that changing existing classes may be the cause of a rise in warnings. In version 22 of SwingWT there is a dramatic rise in warnings. One explanation is that performance of the system at this point started to deteriorate (Section 7.4.5 describes how many of the warnings for this system were generated from potential performance issues).

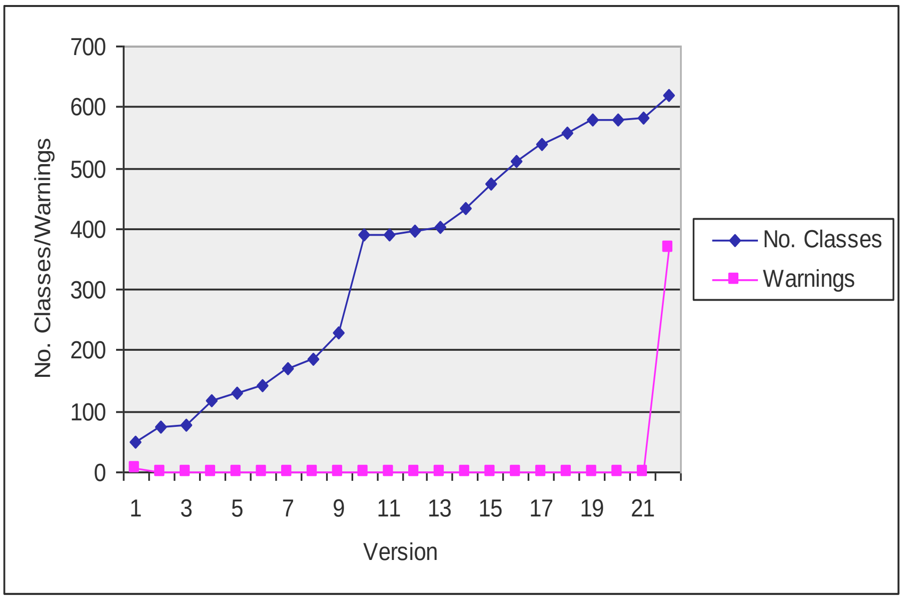


Figure 7.3. Classes/Warnings (SwingWT)

The DIT pattern for the JasperReports system is shown in Figure 7.4 (the figure is also presented in Chapter 3 (Figure 3.2)) and Figure 7.5 shows the trend in number of system classes and warnings generated across versions for JasperReports. No obvious trend between the DIT values and warnings is evident, although both rise gradually over the course of the versions studied. One feature not evident from the figure is that at version 8, the rise in the number of classes and warnings was the largest for both variables. From version 1 to 2, a small decrease in the number of classes resulted in a correspondingly small decrease in warnings. This suggests a strong correlation between the two variables.

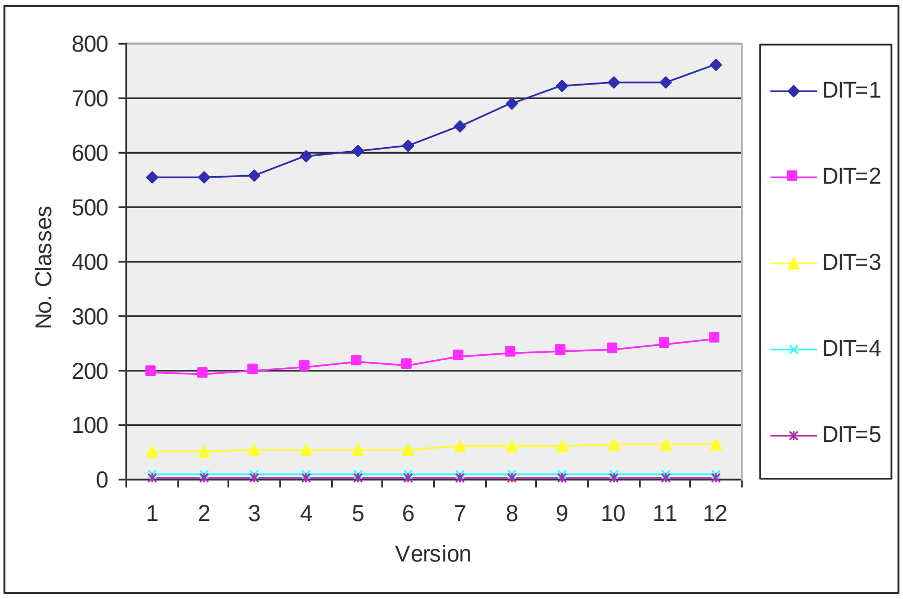


Figure 7.4. DIT frequencies JasperReports (all versions)

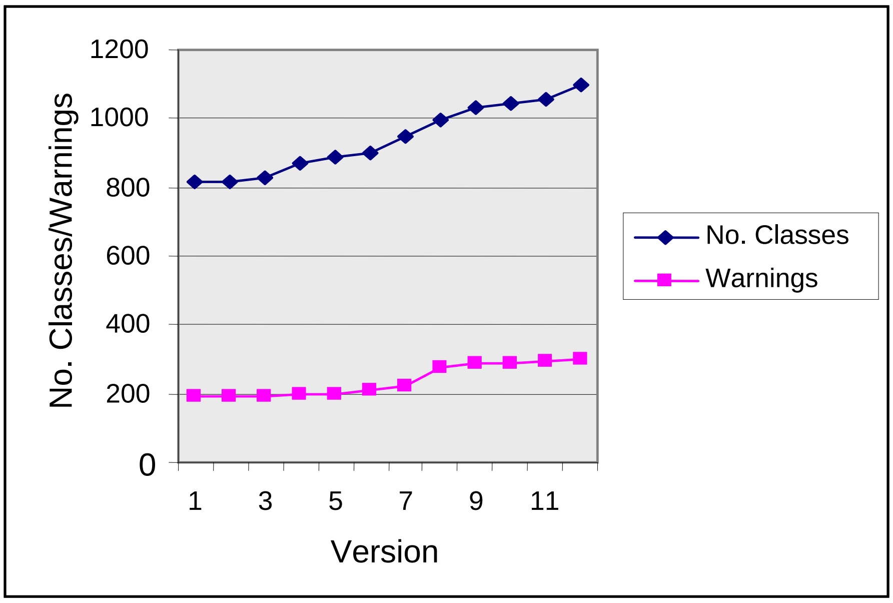


Figure 7.5. Classes/Warnings (JasperReports)

Figure 7.6 shows the DIT frequencies for JColibri and Figure 7.7 shows the evolution of overall classes and warnings in the same system. The trend for JColibri (Figures 7.6 and 7.7) is unlike the trend for the other two systems. We note a strong correspondence between the shape of the graph for DIT level one of Figure 7.6 and the two graphs in Figure 7.7. One particularly noteworthy feature is the decrease in classes and number of warnings in version 8 of the system investigated. One plausible explanation for this trend might be that significant re-engineering and/or refactoring took place at this point in the life of the system and, as a result, some classes were merged and/or deleted. Visual inspection of the classes by the authors revealed removal of significant numbers of *inner classes* from version 7 to version 8. Since inner classes were a source of many warnings in all versions of JColibri (see Section 7.4.5) removal of those classes also accounts for the sudden drop in both classes *and* warnings in Figure 7.7. This was a surprising finding from the analysis. Inner classes could be criticized for adding an extra level of complexity to a class and this may explain their removal.

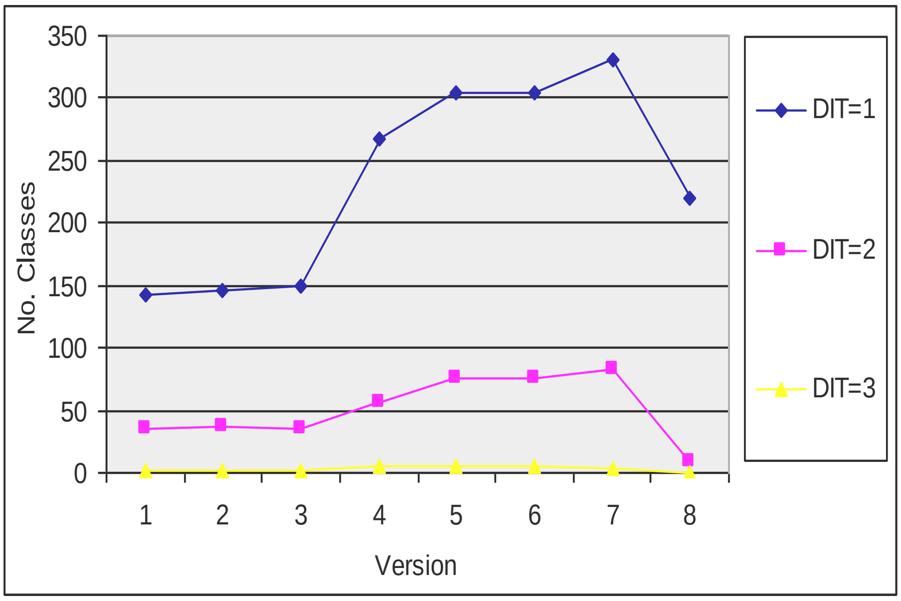


Figure 7.6. DIT frequencies JColibri (all versions)

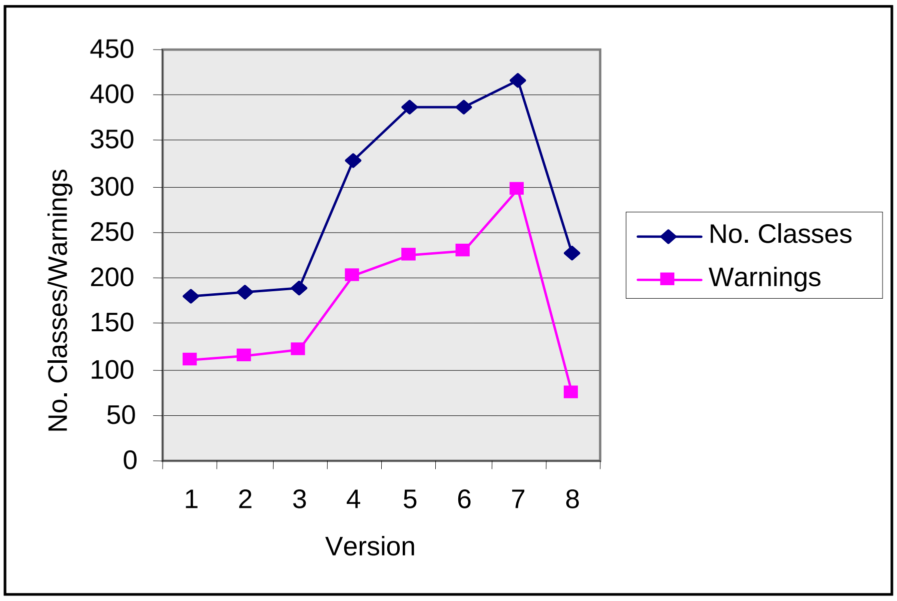


Figure 7.7. Classes/Warnings (JColibri)

Of course, we would expect a certain amount of new classes to be added to a system and for many of those added classes to be at DIT level one; however, the scale of the additions at this level suggests that many of those classes were added without thought as to whether they could fit into a deeper level of the existing hierarchy.

We return to our first research question: *Does the position of a class in inheritance hierarchy, according to its DIT, influence the number of warnings generated as a system evolves*?

In the three systems studied, we observed that the majority of warnings emerged from classes at DIT one where the majority of functionality resided. On the basis of the evidence presented for the three systems, we suggest that while the majority of functionality and warnings do emerge from DIT level one classes, there may be other factors that strongly influence the number of warnings generated by a system (other than simply its position in an inheritance hierarchy). The presence of inner classes alongside significant re-engineering effort are just two facets of a systems' evolution that may influence the propensity for warnings to arise. One argument for why so many warnings were generated for classes at DIT level one may be that such classes are coupled to classes in other more complex ways simply to compensate for the lack of coupling via inheritance.

7.4.4 NOC and Warnings Analysis

As well as analysis of the DIT metric, an equally relevant feature of an inheritance hierarchy is the Number of Children (NOC) metric (Chidamber and Kemerer 1994). This metric provides an indication of the 'breadth' of an inheritance hierarchy rather than its depth as given by the DIT metric. Table 7.6 shows maximum (Max), median (Med), mean (Mn.) and standard deviation (STDEV) NOC values for the final versions of each of the three systems.

Systems	Max	Med	Mn.	STDEV
SwingWT	17	0	0.31	1.31
JasperReports	75	0	0.31	2.80
JColibri	3	0	0.04	0.25

Table 7.6. Summary NOC data for the three systems (final versions)

From Table 7.6, JasperReports contains the highest maximum value for NOC at 75. The NOC values for JColibri indicate that subclassing was used infrequently in this system. (JColibri was found to be the smallest system studied in terms of number of classes.) The median value for all three systems is zero and the low mean NOC values coupled with shallow levels of DIT from our prior analysis suggests that many classes found at DIT one had no children (i.e., subclasses) either. Table 7.7 presents a count of number of classes for each value of the NOC (Classes), the number of warnings in each category (Warning), average number of warnings per class in each category (Warnings/Classes), percentage of classes in each category (Class%) and percentage of warnings in each category (Warnings%) for the final version of SwingWT.

NOC	Classes	Warnings	Warnings/Classes	Class%	Warnings%
0	535	316	0.03	86.29	85.87
1	49	25	0.51	7.90	6.79
2	19	11	0.58	3.06	2.99
3	8	5	0.62	1.29	1.36
4	3	0	0	0.48	0
5	1	1	1	0.16	0.27
6	0	0	0	0	0
7	1	0	0	0.16	0
>7<=17	4	10	2.5	0.65	2.72

Table 7.7. Warnings and NOC in SwingWT (version 22)

From Table 7.7, the number of warnings tends to decrease as NOC increases. The vast majority of classes (i.e., 82.29%) in the SwingWT system have zero NOC and the 535 classes (column 2) account for the 316 of the 368 warnings in total (this represents 85.87% of all warnings). Figure 7.8 shows the pattern of classes (upper graph) and classes with at least 1 warning (lower graph) at the different levels of DIT for classes with NOC=0 in the final version of SwingWT. For example, 385 classes had a DIT one and an NOC=0 and 98 of those 385 classes contained at least one warning. At DIT two, there were 85 classes with NOC=0, of which 20 contained at least one warning.

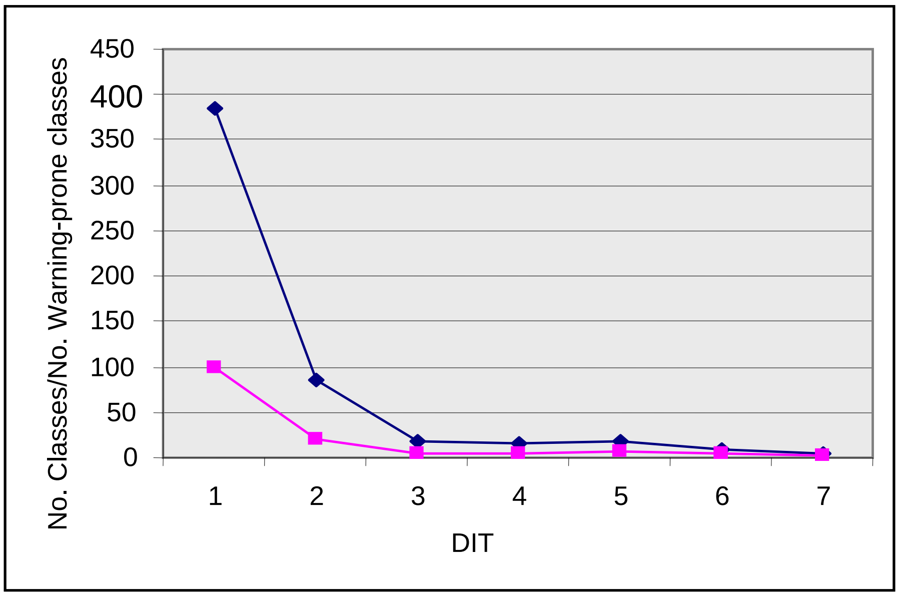


Figure 7.8. DIT levels and Warnings with NOC=0 for SwingWT (version 22)

For the SwingWT system, we thus have a pattern of a relatively large number of classes at DIT one with NOC of 0 and those classes account for a high percentage of warnings. This set of values ties in with the evolutionary pattern for classes to be added to DIT one in Figure 7.2. Figure 7.9 shows the number of warnings and NOC values for version 12 of JasperReports; the rightmost column (NOC > 2) indicates that zero warnings were found in classes with NOC > 2. From Figure 7.9, the highest number of warnings arises from classes with zero NOC.

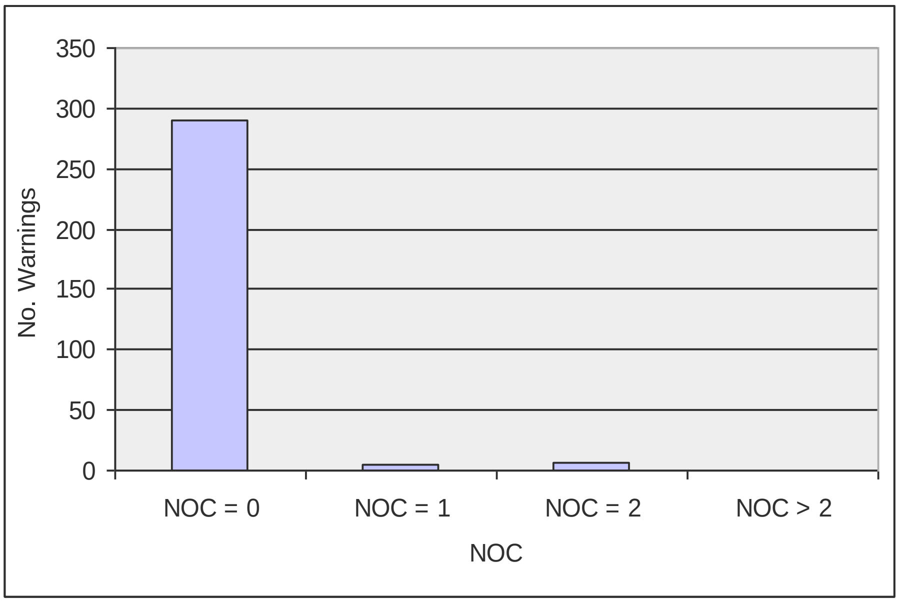


Figure 7.9. Warnings and NOC in JasperReports (version 12)

Table 7.8 presents the data for JasperReports. Just as for SwingWT (Table 7.7), we note high numbers of classes with zero NOC. Inspection of the raw data revealed that 701 of the 1007 classes at DIT level one and a further 233 classes at DIT level two also had an NOC of 0.

NOC	Classes	Warnings	Warnings/classes	Class%	Warnings%
0	1007	290	0.30	91.71	96.34
1	36	5	0.14	3.27	1.66
2	22	6	0.27	2.00	1.99
3	12	0	0	1.09	0
4	10	0	0	0.91	0
5	2	0	0	0.18	0
6	4	0	0	0.36	0
7	2	0	0	0.18	0
>7<=75	3	0	0	0.27	0

Table 7.8. Warnings and NOC in JasperReports (version 12)

Table 7.9 presents the data for the final version of JColibri. In common with both SwingWT (Table 7.7) and JasperReports (Table 7.8) a high percentage of classes (214 from 222) at DIT one had an NOC of 0. Our earlier suggestion that 'large-scale avoidance of deep inheritance levels in a system will cause potential warnings to be found in classes with relatively low NOC values' does not seem to have entirely been borne out for at least one of the systems studied (i.e., SwingWT), although classes at DIT one *and* with zero children seem to figure prominently in warnings even for this system.

NOC	Classes	Warnings	Warnings/classes	Class%	Warnings%
0	222	72	0.08	97.37	97.30
1	5	1	0.20	2.19	1.35
2	0	0	0	0	0
3	1	1	1	0.44	1.35

Table 7.9. Warnings and NOC in JColibri (version 8)

The original research question with respect to NOC was: *Does the number of subclasses of a class, according to its NOC, influence the number of warnings generated in that class*?

In answer to this question, we could suggest that classes with an NOC of 0 house the overwhelming majority of warnings in each of the systems studied. One plausible explanation for this can be that a large number of classes in the three systems contained 0

NOC. The emergence of warnings were therefore due to the large number of classes falling in that category (NOC = 0).

7.4.5 Warnings and refactoring analysis

The third research question stated in Section 7.3.3 related to the possibility of refactoring code on the basis of warning data *as a system evolves*. A previous study by Counsell et al. (Counsell et al. 2006a) has shown that inheritance-related refactorings (e.g., 'Extract Subclass' and 'Extract Superclass') are rarely undertaken in Java systems and the complexity associated with those refactorings may be a contributing factor. On the other hand, frequent occurrences of simple renaming of fields, renaming of methods and movement of fields and methods between classes were observed in the same study. The potential benefit of exploring the research questions is that if sufficient warning signals of 'code smells' for example (Fowler 1999) can be provided at any stage in a system's evolution, in our case through code warnings; then this may help to solve an ongoing question for developers and researchers which is 'what do we refactor and when do we refactor?

Table 7.10 shows the number and type of warnings for the final versions of each system. From Table 7.10, 159 (i.e., 43.2%) warnings in SwingWT belong to the performance (PF) category. However, only 9 (i.e., 2.99%) and 19 (i.e., 24.67%) of the same category of warnings were found in JasperReports and JColibri, respectively, where the Bad Practice (BP) category figured most prominently.

Systems	BP	CO	MCV	MTC	PF	DG
SwingWT	64	48	42	5	159	50
JasperReports	154	18	109	4	9	7
JColibri	27	4	4	1	19	19

Table 7.10. The categories of warnings

We first inspect the data for the SwingWT for the PF category since this is where the highest number of warnings were found and illustrate how those warnings suggest opportunities for potential refactorings. For this system, four of the six PF warnings at version one were the ‘Unread Field (UF)’ warning. The description of the warning taken from (FindBugs 2008) is ‘*This field is never read. Consider removing it from the class*’. If the field in question was assigned from a parameter passed in to the body of the methods, then we could easily apply the ‘Remove Parameter’ (Fowler 1999) refactoring to each method to remove the parameter *and* the field in question; all four of these warnings related to classes at DIT level two. One of the other warnings was also directly related to inheritance, namely: ‘*Class defines field that masks a superclass field (CMS)*’. The description of the warning is: ‘*This class defines a field with the same name as a visible instance field in a superclass. This is confusing, and may indicate an error if methods update or access one of the fields when they wanted the other*’. This problem can be solved easily by a simple ‘Rename Field’ refactoring. The SwingWT system only exhibited warnings at versions one, two and twenty-two. At version 22, the same two warnings re-appeared multiple times. We found 19 occurrences of UF warnings and 30 occurrences of the CMS warning, many of which were at DIT levels three, four, five and six. We also found evidence of ‘unused’ fields in the same set of warnings which, although not strictly relating to any recognized refactoring, could be removed as part of an optimization process.

By far the most common performance warning related to the warning: ‘*Should be a static inner class’*. The description of this warning taken from (FindBugs 2008) is: ‘*This class is an inner class, but does not use its embedded reference to the object which created it. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made static’*. 42 occurrences of this warning were found occurring at all levels of the inheritance hierarchy. A recognized refactoring is to take an outer class and make it a static inner class of another class (if a class is inner, it should be made static anyway). Evidence of confusing method names and unconventional naming of classes was also found by the tool - again these fell into the category of simple renaming refactorings. From a refactoring perspective, the SwingWT system therefore provides ample opportunity at all levels of the inheritance hierarchy for application of relatively easily applied refactorings. The fact that inner classes repeatedly figure in our analysis of the systems studied implies that they are not as convenient and useful as they might suggest.

For JasperReports the most commonly recurring warning in this system was for the Bad Practice Category. Two types of warning dominated the versions of JasperReports:

1. *'May expose internal representation by returning reference to mutable object'*. The description of this warning taken from (FindBugs 2008) is: '*returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by entrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better in many situations'*.
2. *'Non-transient, non-serializable instance field in serializable class'*. The description of this warning taken from (FindBugs 2008) is: '*This Serializable class defines a non-primitive instance field which is neither transient, Serializable, or java.lang.Object, and does not appear to implement the Externalizable interface or the readObject() and writeObject() methods. Objects of this class will not be de-serialized correctly if a non-Serializable object is stored in this field'*.

For warning 1, 49 occurrences were found in version 1 and 60 occurrences of the same warning were found in version 12. A principle upon which the refactoring process rests is that wherever possible, data should be made immutable to prevent its accidental modification *and* to limit the amount of re-test after refactoring. In other words, there is an opportunity for adhering to sound refactoring practice in JasperReports by inspecting each occurrence of these warnings and resolving that potential problem. For warning 2, 122 occurrences were found in the version 1 of this system and 119 occurrences found in version 12. Appropriate definition of fields particularly those involved in Java remote calls is an important part of 'binary refactoring' a technique to improve program performance in which refactoring is achieved without modifying the source code (Tilevich and Smaragdakis 2005). In other words, while there is no specific refactoring that can be applied in response to warning 2, the effectiveness and possibility of applying other refactorings may be harmed if these warnings are allowed to remain. Indeed, there seems only limited evidence that this warning was heeded across the versions of JasperReports.

For JColibri, the most commonly recurring warning was in the Bad Practice category (27 occurrences were found by the tool). The single most common warning in that category was: '*Comparison of String parameter using == or !=*'. The description of this warning taken from (FindBugs 2008) is: *'This code compares a `java.lang.String` parameter for reference equality using the == or != operators. Requiring callers to pass only String constants or interned strings to a method is unnecessarily fragile, and rarely leads to measurable performance gains. Consider using the `equals(Object)` method instead'.*

In version 1 of this system, 48 warnings of this type were generated (from a total of 111) – all of those 48 warnings were for classes located at DIT level one and this suggests that this is code typically inherited by subclasses. In version 8, zero of the 48 warnings were generated. Related to refactoring, improper use of the string comparison techniques is widely considered to be a *bad smell* in code (Fowler 1999); such a smell can be eliminated through application of relatively simple refactoring principles and it appears that smell eradication is what happened in this case to the 48 occurrences of this warning. Further evidence that these refactorings had taken place was the existence of warnings related to '*equals(Object)*' code in a later version. As well as this warning, there were frequent occurrences of the '*Method names should start with a lower-case letter*' warning (these can easily be remedied by simple renaming refactorings) and, in common with the SwingWT system, regular occurrence of the: '*Should be a static inner class'* warning. It was noteworthy that the same types of warning recurred across all three systems but with different emphases.

Investigation of the warnings in the three systems thus demonstrates ample opportunity for refactoring of different types. More generally, if we now return to the third research question of Section 7.3.3: *Does the type of warnings, extracted by FindBugs, inform any refactoring effort that could be applied to the classes as they evolve*?

We suggest that there is definite evidence that not only is refactoring plausible and a practical reality across versions, but has actually been carried out by developers in certain cases.

7.5 Discussion

There are a number of issues related to the study that impact on its validity and/or generalizability. Firstly, the study could be criticised for using the systems with different number of versions (SwingWT containing twenty-two versions, JasperReports containing twelve versions and JColibri containing eight versions). In defence of this criticism, we opted to use the single (final versions) and all available versions of each system. Secondly, we have suggested that warnings tend to occur at DIT one and this is a situation that should have been avoided. However, we have no data to suggest that any other configuration of classes would be any better in terms of the number of warnings generated. In other words, developers add classes at DIT one because that is the sensible thing to do.

Finally, our study may be criticised for using warnings rather than using actual fault data. In our defence, while ideally we would have liked to base our analysis on actual fault data obtained through running and testing the code, we argue that the process of testing OSS is different to that of proprietary system. Obtaining detailed information on each fault from OSS (i.e., which class a particular fault emerges from and how each fault has been remedied) is infeasible. We therefore, opted to use the FindBugs tool to highlight the areas that potentially store problems which may lead to faults in future releases of a system.

7.6 Summary

In this chapter, we empirically investigated the influence of inheritance on warnings issued by an automated tool. Warning and inheritance data was extracted from three Java OSS using the JHawk and FindBugs tools. A number of results emerged from the analysis. Firstly, using just the warnings generated by the FindBugs tool allowed us to model trends in inheritance and to establish whether any relationship existed between how an inheritance hierarchy evolves and the propensity for warnings. Secondly, it demonstrated how generated warnings can be used to inform refactoring effort by pointing to potential hotspots in code (before actual faults appear). In the three systems studied, we found that the majority of warnings were found where the majority of functionality resided (classes at DIT level one and with NOC = 0). In addition, we found that the types of warnings could help identify refactoring effort that could be applied to the classes of an evolving system. Our study also shows that many of the simple, common refactorings proliferate in code and can be highlighted using a simple automated tool. Finally, the analysis is of relevance to developers interested in identifying where remedial action may be required from an inheritance/refactoring perspective.

CHAPTER 8 Conclusions and Future Work

In this chapter we describe the findings and achievements of the research presented in this Thesis. In Section 8.1, we discuss our findings with a reflection on our original objectives presented in Chapter 1. We also describe how those objectives were achieved reflecting on studies presented in each chapter. Section 8.2 provides a description of our contribution. In Section 8.3, we give a description of our personal achievement for undertaking the research in this Thesis and finally, in Section 8.4 we point to possible future work that the research in this Thesis may lead to.

8.1 Thesis Objectives Re-visited

For the research in this Thesis the following objectives, originally stated in Chapter 1, were formed:

1. To improve our understanding of inheritance in Java OSS. That is, to obtain a greater understanding of what inheritance is and how this OO mechanism is used in practice.
2. To investigate quantitatively how inheritance hierarchies evolve as opposed to that of system evolution as a whole. In particular, to conduct a thorough investigation of where (in the inheritance hierarchy) the majority of incremental changes are applied as a system evolves.
3. To investigate evolution of inheritance from a classes movement and relocation perspective. In other words, to investigate how classes within an inheritance hierarchy are moved from one level to another as a system evolves.
4. To investigate inheritance from a class interaction perspective.

In the following, we demonstrate how we addressed these objectives referring to the empirical evidence presented throughout the Thesis, and how those findings can build a body of knowledge in the domain of OO.

To address the aforementioned objectives, we started our investigation of inheritance by conducting a thorough literature review (see Chapter 2) of previous work reported on inheritance, how it was used in practice and what implications it may have on system maintainability. This followed with the study presented in Chapter 3, which was concerned about how incremental changes were made to inheritance hierarchies as seven Java OSS evolved (the class changes were investigated in seven systems and the method and attributes changes were explored in three systems). That is, we investigated where, within the inheritance hierarchy, the majority of changes (class, method and attribute additions/deletions) were made as the systems evolved from one version to the next. In Chapter 4 we investigated how inheritance hierarchies, in the remaining four systems, changed and those changes were analogized to a set of low-level refactorings (method and attribute-related refactoring, Fowler (1999)) applied to initial versions of the systems; the refactoring data was available for only initial versions of the systems.

Chapters 2, 3 and 4 therefore played a significant part in establishing an understanding of what inheritance is, people's (other researchers) views of inheritance from a maintainability perspective, how inheritance evolves in terms of incremental changes at various levels of granularity (i.e., class, method and attribute). Chapter 5 described an investigation of inheritance evolution from a perspective of class movement and relocation, as well as class addition and deletion in a selected set of versions of four OSS. The analysis of class movement and relocation added an additional level of rigour to our investigation of inheritance. By analyzing merely incremental class changes we would have missed the class movement within inheritance hierarchy. Class movement and relocation is a system restructuring activity which is often undertaken during the system reengineering. Our claim that class movement and relocation improves software comprehensibility which is an aid to software maintenance; reflects previous findings on software restructuring/refcactoring (Johnson and Foote 1988, Chikofsky et al. 1990).

In Chapter 6 we explored class interaction and its evolution taking inheritance as the main unit of our analysis in four Java OSS. In other words, we presented a study investigating where, in an inheritance hierarchy, classes are predisposed to calling methods of other classes (calls to methods of the classes within the line of hierarchy to the root and to classes outside the line of hierarchy), and how this class interaction evolved. The findings also suggested that classes with method invocations, either/both to classes within the line of hierarchy or those outside the line of hierarchy, have low cohesion. The study provided an insight into how class interaction patterns within and across an inheritance hierarchy and how that class interaction evolved.

In Chapter 7 we investigated the impact of inheritance on warnings (problems embedded in the system which may lead to faults) extracted by the FindBugs tool (FindBugs 2008). In the same chapter, we also investigated the refactoring opportunities that those extracted warnings yielded. The study in Chapter 7 gave some interesting insights into warnings and potential refactorings.

Based on the results of empirical studies presented throughout the Thesis, we therefore feel that all 4 objectives stated in Chapter 1 have been satisfied. The trends of inheritance at various levels of granularity in the evolution of Java OSS presented interesting characteristics. We therefore assert that the Thesis informs our empirical understanding of inheritance feature of OO from an evolutionary perspective.

8.2 Contribution

As stated in Chapter 1, the contribution of this Thesis in the realm of SE can be justified on the following basis. We found no other prior empirical studies to investigate inheritance from the perspective of evolution, despite the fact that it is considered as a prominent OO feature. Equally, empirical evidence exists to suggest that research on software evolution is conducted inadequately. A need for further empirical studies, in particular, at various levels of granularity, has been stressed (Kemerer and Slaughter 1999).

The main contribution of this Thesis can be seen in the light of three research stands. Firstly, an appreciation of trends of inheritance can help predict future changes in an inheritance hierarchy. Secondly, based on the trends of changes, developers can take pre-emptive actions for further system maintenance and/or refactorings. Finally, since no empirical study to date has analyzed inheritance from an evolutionary perspective at various levels of granularity, we believe the methodological approaches adopted for data collection and analysis in this Thesis can help inform future empirical studies on inheritance and its evolution. The Thesis therefore makes a contribution to our understanding of how inheritance hierarchies evolve and where the majority of maintenance changes are applied. The findings of Chapter 3 suggested that approximately 96% of overall maintenance changes (addition and deletion of classes) were made at level one and two of class hierarchy. Only 4% of the same changes were made at and beyond level three. In terms of methods and attributes, the evidence suggested that approximately, 93% of method changes and 97% of attribute changes were made at level one and two of inheritance hierarchy, the remaining changes were made at level three and beyond. This was a surprising result to emerge from our investigation, suggesting that developers of the systems paid far too much attention at levels one and two of inheritance hierarchies while there might have been opportunities for making such maintenance changes at lower levels (levels three and beyond). The findings of Chapter 4 suggested that analyzing systems at lower level of granularity (i.e., method and attribute level) can often show a different trend to that of a similar analysis of the system at a higher grain (i.e., class and packages level).

In addition to incremental changes in an inheritance hierarchy, the Thesis also contributed to a body of knowledge of how classes were moved within inheritance hierarchies. The empirical findings in Chapter 5 suggested that in the set of OSS studied, 1) larger classes and 2) tightly classes were more frequently moved within inheritance hierarchies than their counterparts (smaller classes and loosely coupled classes). The findings also indicated that larger classes and tightly coupled classes were less cohesive than smaller classes and loosely coupled classes, respectively, and the lack of cohesion in those classes may have influenced the decision to move them within their respective class hierarchies. The evidence in Chapter 5 also indicated that, in line with other empirical evidence (Daly et al. 1996), a maximum of three levels of inheritance *may be* a preferred amount (from a depth perspective) to be used in a system.

Again, we have found no prior empirical studies that have investigated class interaction within and across inheritance hierarchy and its evolution. The findings in Chapter 6 suggested that the majority of method invocations were made to methods of the classes where the majority of functionality existed (levels one and two of class hierarchy), and there is a positive correlation between number of methods in a class and calls to methods of other classes. From a fault perspective, despite the fact that previous studies (Briand et al. 2000, Briand et al. 2001, Briand et al. 2002, Cartwright and Shepperd 2000) have investigated the role and impact of inheritance on fault-proneness, there is still a conflict of views on inheritance in relation to fault-proneness. In Chapter 7 the warnings extracted by FingBugs tool (FindBugs 2008) as a replacement to actual faults was explored. The empirical evidence in Chapter 7 indicated that the majority of warnings were found where the majority of functionality existed (levels one and two of class hierarchy), irrespective of class position in the inheritance hierarchy. The evidence also suggested that the warnings generated by FindBugs could highlight potential hotspots before actual faults appear, and those warnings could be used to target potential refactorings in those systems.

8.3 Personal Achievement

There are numerous things that I have achieved over the course of the research in this Thesis. I firstly, learned how empirical research is conducted and what makes good research. I also acquired an appreciation of the difficulties associated with the process of conducting sound research. That is, research is a challenging activity.

Time management is a key aspect of research. Good time management can help meet deadlines. In addition, research collaboration and communication is an important aspect of successful research. During the course of this Thesis, I communicated and when possible collaborated with colleagues within the university (Brunel University UK) and across the world. This included communicating my research with other researchers when attending conferences and also collaborating in studies with some colleagues when possible (see list of publications for the collaborative work).

My achievement also included an understanding of characteristic of the process of undertaking research. For example, research should be thorough and focused in a small and narrow area in order to make a contribution in the area of interest, rather than tackling a wider area superficially. The process of conducting research also includes a need for an advancement of the researcher’s knowledge in the problem area, obtaining other researchers' views on the subject area and critical assessment of prior work carried out on the research topic.

The research in this Thesis required a certain amount of data collection and statistical analysis which helped me improve my understanding of data collection and appropriateness of statistical tests for a set of data. Over the course of this Thesis, I also learned that completing a PhD is only a learning process for further scientific research. It helps acquire knowledge and expertise on how research is conducted, the *ups* and *downs* of academic life and most importantly how to be a scholarly researcher post-completion of the PhD.

8.4 Future Work

Since inheritance evolution has received little attention, if any, from the SE research community, there are several research topics that this Thesis could lead to.

In Chapter 3 we investigated the trends in the additions and deletions of classes within inheritance hierarchies in seven Java OSS and the changes of methods and attributes in three of the seven OSS. Future work in this particular area will be to use various lower level measures (i.e., lines of commented/non-commented code, number of statements) to investigate evolutionary and maintenance forces of inheritance in the studied systems. This will be of importance for two reasons, firstly, to capture lower-level change pattern and secondly, to compare those lower-level changes to the changes of inheritance at class, method and attribute level.

In Chapter 5 we manually investigated the movement of classes within inheritance hierarchies in four Java OSS which could lead to a number of studies. Our results showed that larger classes and highly coupled classes were more frequently moved within inheritance hierarchies in the four systems. We firstly, plan to build a prediction model to predict class movements within an inheritance hierarchy based on size, coupling and cohesion. Another avenue of future work in this area would be to investigate high-level refactorings (refactorings relating to restructuring of class hierarchy) carried out based on the class movements in those four systems. A third future research topic will be to investigate the movement of methods and attributes within inheritance hierarchies in those systems to inform our understanding of how methods and attributes moved in contrast to movement of classes in those four systems.

Another area of future research is to investigate the impact of inheritance on actual faults experienced in a system. In Chapter 7 we investigated this phenomenon using the warnings, since actual fault data for OSS used was not available. While our analysis presented some interesting results, we believe analyzing actual faults would have provided further insight into the systems and their maintenance.

Another area of future work would be to employ various research methods to ascertain the impact of inheritance on maintainability and its evolution. For example, it would be interesting to carry out a qualitative analysis of inheritance using interviews and/or questionnaires to obtain views of experts in academia and industry on inheritance. A further avenue of future research would be to replicate the studies carried out in this Thesis using proprietary systems. Throughout this Thesis we used eight OSS systems as a testbed. While the OSS systems used herein are equivalent, from a size and functionality perspective, to that of industrial systems, we believe it would be interesting to compare the results of the studies on inheritance obtained from both OSS and proprietary systems and draw further generalizable conclusions.

Appendix A: Glossary of Software Engineering Terms

The terms define below are ubiquitous 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.

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.

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.

Inner class

In Java an inner class is a class defined within another class or method and have access to its enclosing class.

Class Member

Class members are attributes and methods defined in a class.

Interface

An interface is a collection of well defined members (method signature) with empty bodies. Interfaces are implemented by classes which provide body to the methods defined in the interface. Interfaces cannot be instantiated.

Abstract Class

An abstract class is a class which defines methods and attributes and is declared as abstract. Abstract classes cannot be instantiated. The difference between an abstract class and interface is that all attributes defined in an interface must be static final and the methods should always be abstract, whereas these criteria do not hold for an abstract class.

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.

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.

Attribute

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

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.

Method Overriding

Method overriding is an OO feature which allows a subclass to have a specific implementation of a method defined in its superclass(s). An overridden method must have the same name and list of argument types as the default method.

Method Overloading

Method overloading is a special type of method overriding which allows a different numbers and/or types of arguments to that of default method.

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 Method (RM) Refactoring

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

Rename Field (RF) Refactoring

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

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).

Pull Up Field (PUP) Refactoring

PUF refactoring moves a field defined in two or more subclasses to their superclass (Fowler 1999). PUF eliminates code redundancy in a system.

Push Down Field (PDF) Refactoring

PDF refactoring moves a field which is not used in all subclasses, to the subclasses(s) in which it is used (Fowler 1999).

Pull Up Method (PUM) Refactoring

PUM refactoring moves a method defined in two or more subclasses, to their superclass (Fowler 1999). PUM also eliminates code redundancy in a system.

Push Down Method (PDM) Refactoring

PDM refactoring moves a method, which is not used by all subclasses, to the subclass in which it is used (Fowler 1999)**.**

Code Smell

A code smell is an indication of a problem in the code which can be eliminated by refactoring. This includes *Duplicate Code*: when a piece of code is detected in more than one place (Fowler 1999). This code smell suggests a number of refactorings depending on the system and the duplicated code.

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.

Dependent Classes

Two classes are dependent on each other when there is coupling between them.

Cohesion

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

Encapsulation

Encapsulation also known as information hiding is an important feature of OO which is concerned with the visibility or accessibility of elements (methods and attributes) of a class to other classes.

Method Call/Invocation

Method call/invocation is the calls to a method define in the same class or a different class.

Software Metrics

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

Depth of Inheritance Tree (DIT) Metric

DIT is a measure of depth of a class in an inheritance hierarchy from the class to the root. In other words, it specifies the position of a class in an inheritance hierarchy from a depth perspective. In Java every class inherits from Object hence the DIT value from a class not inheriting from any other class is 1.

Number of Children (NOC) Metric

NOC measures the number of immediate subclasses of a class.

Specialization Ratio (SR) Metric

SR measures the number of subclasses of a class divided by the number of its superclasses. High values of the SR metric imply high level of reuse through subclassing.

Reuse Ratio (RR) Metric

RR metric measures the number of superclasses of a class divided by the total number of classes. The total number of classes refers to total number of classes residing in an inheritance hierarchy excluding class 'Object'.

Number of Methods (NOM) Metric

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

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.

Calls to methods within Hierarchy (HIER) Metric

HIER metric measures the number of method calls that are in class hierarchy for a class.

Number of External methods calls (EXT) Metric

EXT metric measures the number of method calls in a class to methods of other classes excluding HIER calls.

Lack of Cohesion Of the Methods in a class (LCOM) Metric

LCOM metric measures the relations of methods and local variables of a class by counting the number of method pairs accessing different fields/variables minus the number of method pairs accessing the same fields/variables.

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.

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.

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 B: Raw Data on Number of Classes from the Eight Systems

The tables in this appendix provide some details on number of classes at each DIT level for the eight systems. For example, from Table B.1, in version 1 of HSQLDB the maximum DIT was two, 54 classes were residing at DIT one and only two classes were found at DIT two.

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	54	2	0	0	56
2	104	23	3	0	130
3	114	29	4	0	147
4	247	63	12	1	323
5	246	64	12	1	323
6	279	68	11	0	358

Table B.1. Number of classes at each DIT level for HSQLDB

Version	DIT=1	DIT=2	DIT=3	Total
1	142	36	1	179
2	146	37	1	184
3	150	36	2	188
4	267	56	5	328
5	304	76	6	386
6	305	76	6	387
7	331	83	3	417
8	220	8	0	228

Table B.2. Number of classes at each DIT level for JColibri

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	556	196	52	10	4	818
2	554	195	52	10	4	815
3	557	201	54	10	4	826
4	593	206	56	10	4	869
5	602	215	56	10	4	887
6	613	209	56	10	4	892
7	648	227	61	10	4	950
8	691	231	61	10	4	997
9	724	234	62	10	4	1034
10	729	240	63	10	4	1046
11	728	249	63	10	4	1054
12	761	258	65	10	4	1098

Table B.3. Number of classes at each DIT level for JasperReports

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	158	20	4	1	183
2	158	20	4	1	183
3	139	21	4	1	165
4	138	21	4	1	164
5	139	21	4	1	165
6	139	21	4	1	165
7	139	21	4	1	165
8	137	21	4	1	163
9	144	22	4	1	171
10	144	22	4	1	171
11	144	22	5	1	172
12	144	23	5	1	173
13	145	23	5	1	174
14	145	23	5	1	174
15	161	23	5	1	190
16	169	23	5	1	198
17	168	24	5	1	198
18	168	24	5	1	198
19	168	24	5	1	198
20	168	24	5	1	198
21	167	24	5	1	197

Table B.4. Number of classes at each DIT level for EasyWay

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
1	29	16	5	0	0	0	0	50
2	46	21	8	0	0	0	0	75
3	47	22	8	0	0	0	0	77
4	66	26	23	4	0	0	0	119
5	70	33	24	4	0	0	0	131
6	80	20	27	12	4	0	0	143
7	102	23	26	11	5	2	0	169
8	111	24	22	18	5	2	3	185
9	143	35	22	20	5	2	3	230
10	263	63	23	22	10	6	3	390
11	263	63	23	22	10	6	3	390
12	272	65	22	21	6	6	3	395
13	275	64	23	24	6	6	4	402
14	296	69	22	25	12	7	2	433
15	329	73	25	27	12	7	2	475
16	359	80	15	23	19	12	4	512
17	380	87	19	25	17	8	2	538
18	399	88	19	25	17	8	2	558
19	406	87	18	25	28	12	3	579
20	406	87	18	25	28	12	3	579
21	410	87	18	25	28	12	3	583
22	429	99	24	25	28	11	4	620

Table B.5. Number of classes at each DIT level for SwingWT

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	112	15	7	3	137
2	100	15	7	3	125
3	100	15	7	3	125
4	100	15	7	3	125
5	103	15	7	3	128
6	109	12	7	3	131
7	112	12	7	3	134
8	113	12	7	3	135
9	114	12	7	3	136
10	115	12	7	3	137
11	117	12	7	3	139
12	117	12	7	3	139
13	113	12	7	3	135
14	113	12	7	3	135
15	113	12	7	3	135
16	112	12	7	3	134
17	113	12	7	3	135
18	113	12	7	3	135
19	113	12	7	3	135
20	113	12	7	3	135
21	113	12	7	3	135
22	114	12	7	3	136
23	114	12	7	3	136

Table B.6. Number of classes at each DIT level for JAG

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
8	3141	599	155	33	4	1	1	3934
9	3152	599	155	33	4	1	1	3945
10	3223	622	156	33	4	1	1	4040
11	3499	645	172	34	4	1	1	4356
12	3247	629	158	33	4	1	1	4073
13	3771	683	178	38	4	1	1	4676
14	3790	689	179	38	7	1	1	4705
15	4176	693	168	23	7	1	1	5069
16	4195	690	168	23	7	1	1	5085
17	5942	1000	290	38	16	4	1	7291
18	4764	837	239	37	17	4	1	5899
19	6230	1046	293	39	16	4	1	7629
20	6259	1046	293	38	16	4	1	7657
21	4896	859	240	37	17	4	1	6054
22	6021	972	263	42	21	9	2	7330
23	7524	1152	378	101	29	12	2	9198
24	7468	1150	391	82	22	8	2	9123
25	7473	1149	392	82	22	8	2	9128
26	4643	809	251	69	15	5	1	5793
27	8536	1312	374	76	22	8	2	10330
28	4689	814	252	59	8	1	1	5824
29	4697	816	249	59	8	1	1	5831
30	8511	1285	367	73	22	8	2	10268
31	7715	1268	288	52	6	1	1	9331
32	7902	1319	295	51	6	1	1	9575
33	8545	1502	333	52	5	1	1	10439
34	7379	1345	295	55	6	1	1	9082

Table B.7. Number of classes at each DIT level for JBoss

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	41	34	22	22	3	122
2	45	41	23	25	5	139
3	45	41	23	25	5	139
4	45	42	22	29	5	143
5	96	13	63	0	0	172
6	96	13	63	0	0	172
7	96	13	63	0	0	172
8	96	13	63	0	0	172
9	96	13	63	0	0	172
10	96	13	63	0	0	172
11	96	13	63	0	0	172
12	96	13	63	0	0	172
13	96	13	63	0	0	172
14	96	13	63	0	0	172
15	96	13	63	0	0	172
16	96	13	63	0	0	172
17	96	13	63	0	0	172
18	96	13	63	0	0	172
19	96	13	63	0	0	172
20	96	13	63	0	0	172
21	96	13	63	0	0	172
22	96	13	63	0	0	172
23	96	13	63	0	0	172
24	96	13	63	0	0	172
25	96	13	64	0	0	173
26	96	13	64	0	0	173
27	101	26	65	0	0	192
28	103	31	65	0	0	199
29	103	33	65	0	0	201
30	109	36	66	0	0	211
31	109	36	66	0	0	211
32	112	37	66	0	0	215
33	112	40	66	0	0	218
34	113	40	66	0	0	219
35	112	41	66	0	0	219
36	125	39	67	0	0	231
37	139	41	66	0	0	246
38	141	42	69	0	0	252
39	141	42	69	0	0	252
40	143	47	79	0	0	269
41	145	47	81	0	0	273
42	146	47	81	0	0	274
43	143	47	81	0	0	271
44	143	47	81	0	0	271
45	142	49	82	0	0	273

Table B.8. Number of classes at each DIT level for Tyrant

Appendix C: Raw Data on number of methods from the Eight Systems

The tables in this appendix provide some details on number of methods at each DIT level for the eight systems. For example, from Table C.1, in version 1 of HSQLDB, 872 methods were residing at DIT 1 and 100 methods were found at DIT 2.

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	872	100	0	0	972
2	1588	314	17	0	1919
3	1651	405	29	0	2085
4	3231	749	178	0	4158
5	3224	756	178	1	4159
6	3800	856	171	0	4827

Table C.1. Number of methods at each DIT level for HSQLDB

Version	DIT=1	DIT=2	DIT=3	Total
1	986	167	2	1155
2	1005	170	1	1176
3	1025	169	5	1199
4	1674	299	17	1990
5	2078	405	29	2512
6	1959	484	46	2489
7	2325	382	30	2737
8	1231	21	0	1252

Table C.2. Number of methods at each DIT level for JColibri

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	4502	2596	1079	20	4	4502
2	4525	2590	1079	20	4	4525
3	4541	2615	1081	20	4	4541
4	4823	2647	1083	20	4	4823
5	4842	2693	1085	20	4	4842
6	4947	2750	1087	20	4	4947
7	5229	2815	1158	20	4	5229
8	6033	2844	1163	20	4	6033
9	6228	2853	1167	20	4	6228
10	6261	2887	1169	20	4	6261
11	6283	2946	1173	20	4	6283
12	6531	3044	1137	20	4	6531

Table C.3. Number of methods at each DIT level for JasperReports

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	1066	165	31	4	1266
2	1074	165	31	4	1274
3	879	190	31	4	1104
4	879	190	31	4	1104
5	881	191	31	4	1107
6	883	191	31	4	1109
7	881	191	31	4	1107
8	876	191	31	4	1102
9	917	201	31	4	1153
10	918	201	31	4	1154
11	931	201	32	4	1168
12	946	203	37	4	1190
13	950	204	37	4	1195
14	950	204	37	4	1195
15	1138	204	37	4	1383
16	1207	204	37	4	1452
17	1204	208	37	4	1453
18	1204	208	37	4	1453
19	1204	208	37	4	1453
20	1204	208	37	4	1453
21	1207	208	37	4	1456

Table C.4. Number of methods at each DIT level for EasyWay

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
1	216	110	52	0	0	0	0	378
2	370	201	86	0	0	0	0	657
3	372	220	110	0	0	0	0	702
4	451	243	150	0	0	0	0	844
5	506	282	154	0	0	0	0	942
6	584	114	301	17	0	0	0	1016
7	744	130	375	79	12	0	0	1340
8	839	138	388	279	15	21	5	1685
9	1147	164	407	335	15	22	5	2095
10	2323	512	600	412	88	76	13	4024
11	2264	512	629	443	87	76	13	4024
12	2312	454	620	440	64	76	13	3979
13	2371	544	643	456	64	76	14	4168
14	2623	584	606	537	116	105	6	4577
15	2817	639	702	549	122	112	6	4947
16	3103	690	305	775	297	114	50	5334
17	3334	884	389	826	298	106	44	5881
18	3562	891	385	840	309	106	44	6137
19	3430	855	389	879	662	170	46	6431
20	3706	1085	279	362	719	201	79	6431
21	3474	855	389	901	683	180	46	6528
22	3722	963	432	915	690	149	85	6956

Table C.5. Number of methods at each DIT level for SwingWT

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	1043	99	67	3	1212
2	992	99	67	3	1161
3	1015	99	67	3	1184
4	1020	99	67	3	1189
5	1093	99	67	3	1262
6	1163	69	67	3	1302
7	1167	69	67	3	1306
8	1173	69	67	3	1312
9	1177	69	67	3	1316
10	1207	69	67	3	1346
11	1227	69	67	3	1366
12	1227	69	67	3	1366
13	1227	69	67	3	1366
14	1227	69	67	3	1366
15	1227	69	67	3	1366
16	1250	69	67	3	1389
17	1260	69	67	3	1399
18	1260	69	67	3	1399
19	1284	69	67	3	1423
20	1289	69	67	3	1428
21	1293	69	67	3	1432
22	1316	69	67	3	1455
23	1333	69	67	3	1472

Table C.6. Number of methods at each DIT level for JAG

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
8	22918	3928	977	213	35	1	1	28073
9	23010	3919	980	213	35	1	1	28159
10	23605	4091	1016	213	35	1	1	28962
11	25310	4364	1137	229	35	1	1	31077
12	23709	4111	1078	213	35	1	1	29148
13	27203	4563	117	256	35	1	1	32176
14	27288	4605	1186	257	48	1	1	33386
15	29227	4539	1050	146	29	1	1	34993
16	29340	4533	1050	146	29	1	1	35100
17	42082	6720	2013	232	70	10	1	51128
18	33869	5475	1378	209	72	10	1	41014
19	44515	6995	2022	232	70	10	1	53845
20	35141	5682	1384	210	72	10	1	42500
21	44735	6996	2022	232	70	10	1	54066
22	44003	6538	1895	226	203	48	6	52919
23	51705	7430	2832	574	257	63	6	62867
24	50874	7137	2829	476	210	48	6	61580
25	50851	7138	2829	476	210	48	6	61558
26	33744	5232	1845	470	81	16	1	41389
27	57026	7991	2557	457	210	49	7	68297
28	33992	5233	1780	447	36	1	1	41490
29	34043	5251	1745	447	36	1	1	41524
30	56822	7866	2530	414	210	49	7	67898
31	50977	7536	2002	304	25	1	1	60846
32	51907	8022	2042	304	25	1	1	62302
33	56863	8532	2300	253	18	1	1	67968
34	51338	8035	2060	318	25	1	1	61778

Table C.7. Number of methods at each DIT level for JBoss

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	364	224	198	185	11	982
2	414	279	216	221	16	1146
3	416	280	217	221	16	1150
4	437	299	206	281	16	1239
5	978	286	143	0	0	1407
6	978	286	143	0	0	1407
7	978	286	143	0	0	1407
8	978	286	143	0	0	1407
9	978	286	143	0	0	1407
10	978	286	143	0	0	1407
11	978	286	143	0	0	1407
12	978	286	143	0	0	1407
13	978	286	143	0	0	1407
14	978	286	143	0	0	1407
15	978	286	143	0	0	1407
16	978	286	143	0	0	1407
17	978	286	143	0	0	1407
18	978	286	143	0	0	1407
19	978	286	143	0	0	1407
20	978	286	143	0	0	1407
21	978	286	143	0	0	1407
22	978	286	143	0	0	1407
23	978	286	143	0	0	1407
24	991	287	143	0	0	1421
25	1001	287	144	0	0	1432
26	1003	295	144	0	0	1442
27	1043	373	145	0	0	1561
28	1029	442	144	0	0	1615
29	1045	474	144	0	0	1663
30	1087	480	158	0	0	1725
31	1087	480	158	0	0	1725
32	1112	490	158	0	0	1760
33	1129	507	159	0	0	1795
34	1149	515	159	0	0	1823
35	1156	523	158	0	0	1837
36	1269	521	164	0	0	1954
37	1417	558	185	0	0	2160
38	1435	574	193	0	0	2202
39	1468	583	196	0	0	2247
40	1495	594	216	0	0	2305
41	1502	500	227	0	0	2229
42	1503	603	228	0	0	2334
43	1500	607	229	0	0	2336
44	1511	621	226	0	0	2358
45	1516	631	227	0	0	2374

Table C.8. Number of methods at each DIT level for Tyrant

Appendix D: Raw Data on Number of Attributes from the Eight Systems

The tables in this appendix provide some details on number of attributes at each DIT level for the eight systems. For example, from Table C.1, in version 1 of HSQLDB, 616 attributes were residing at DIT 1 and only 6 attributes were found at DIT 2.

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	616	6	0	0	622
2	993	52	4	0	1048
3	1041	106	8	0	1155
4	2903	246	33	1	3183
5	2902	246	33	1	3182
6	3227	301	20	0	3548

Table D.1. Number of attributes at each DIT level for HSQLDB

Version	DIT=1	DIT=2	DIT=3	Total
1	380	32	0	412
2	407	33	0	440
3	435	25	5	465
4	876	63	9	948
5	953	187	12	1152
6	919	176	17	1112
7	1075	194	6	1275
8	549	8	0	557

Table D.2. Number of attributes at each DIT level for JColibri

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	2426	929	281	4	2	3642
2	2461	927	281	4	2	3675
3	2463	931	282	4	2	3682
4	2631	948	282	4	2	3867
5	2611	962	282	4	2	3861
6	2671	984	282	4	2	3943
7	2765	1016	316	4	2	4103
8	2979	1033	316	4	2	4334
9	3076	1042	319	4	2	4443
10	3109	1022	321	4	2	4458
11	3554	689	322	4	2	4571
12	3700	752	318	4	2	4776

Table D.3. Number of attributes at each DIT level for JasperReports

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	960	43	4	2	1009
2	963	43	4	2	1012
3	834	50	4	2	890
4	834	50	4	2	890
5	834	50	4	2	890
6	835	50	4	2	891
7	834	50	4	2	890
8	835	50	4	2	891
9	865	64	4	2	935
10	865	64	4	2	935
11	867	64	4	2	937
12	877	67	4	2	950
13	879	68	4	2	953
14	879	68	4	2	953
15	1009	68	4	2	1083
16	1049	68	4	2	1123
17	1050	68	4	2	1124
18	1050	68	4	2	1124
19	1050	68	4	2	1124
20	1050	68	4	2	1124
21	1051	68	4	2	1125

Table D.4. Number of attributes at each DIT level for EasyWay

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
1	130	34	100	0	0	0	0	264
2	160	47	19	0	0	0	0	226
3	160	59	27	0	0	0	0	246
4	308	66	37	0	0	0	0	411
5	316	86	37	0	0	0	0	439
6	375	33	87	3	0	0	0	498
7	434	38	102	7	0	0	0	581
8	478	37	107	53	2	5	3	685
9	639	44	108	71	2	6	3	873
10	1036	174	141	111	38	15	3	1518
11	1062	188	149	90	11	15	3	1518
12	1112	190	127	88	41	15	3	1576
13	1159	187	133	90	8	15	3	1595
14	1225	311	126	102	16	24	3	1807
15	1109	272	255	104	16	25	3	1784
16	1180	284	320	182	60	9	9	2044
17	1358	278	336	194	62	16	8	2252
18	1417	285	333	195	62	16	8	2316
19	1294	276	334	195	239	30	8	2376
20	1204	476	342	75	218	41	19	2375
21	1300	276	334	197	248	30	8	2393
22	1436	294	345	197	248	26	13	2559

Table D.5. Number of attributes at each DIT level for SwingWT

Version	DIT=1	DIT=2	DIT=3	DIT=4	Total
1	541	31	37	0	609
2	517	31	37	0	585
3	526	31	37	0	594
4	526	31	37	0	594
5	604	31	37	0	672
6	675	19	37	0	731
7	706	19	37	0	762
8	722	19	37	0	778
9	725	19	37	0	781
10	748	19	37	0	804
11	764	19	37	0	820
12	786	19	37	0	842
13	805	19	37	0	861
14	805	19	37	0	861
15	805	19	37	0	861
16	816	19	37	0	872
17	835	19	37	0	891
18	836	19	37	0	892
19	847	19	37	0	903
20	850	19	37	0	906
21	853	19	37	0	909
22	864	19	37	0	920
23	871	19	37	0	927

Table D.6. Number of attributes at each DIT level for JAG

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6	DIT=7	Total
8	8723	1330	369	87	12	1	0	10522
9	8754	1330	369	87	12	1	0	10553
10	8950	1338	389	87	12	1	0	10777
11	9387	1435	434	97	12	1	0	11366
12	8990	1341	390	87	12	1	0	10821
13	10077	1490	443	106	12	1	0	12129
14	10137	1496	443	104	14	1	0	12195
15	10854	1355	352	52	8	1	0	12622
16	10884	1353	352	52	8	1	0	12650
17	16008	1920	502	53	15	1	0	18499
18	13107	1602	458	61	15	1	0	15244
19	16716	2002	502	53	15	1	0	19289
20	13534	1656	455	61	15	1	0	15722
21	16758	2005	502	53	15	1	0	19334
22	16542	1937	503	55	46	10	0	19093
23	19908	2140	581	75	46	10	0	22760
24	19355	2102	644	67	46	10	0	22224
25	19359	2104	643	67	46	10	0	22229
26	12347	1503	401	67	8	1	0	14327
27	21583	2318	624	62	46	11	0	24644
28	12474	1503	381	63	8	1	0	14430
29	12481	1508	377	63	8	1	0	14438
30	21318	2306	640	56	46	11	1	24378
31	17936	2067	488	39	6	1	0	20537
32	18191	2220	501	39	6	1	0	20958
33	19923	2263	450	38	6	1	0	22681
34	18272	2205	503	39	6	1	0	21026

Table D.7. Number of attributes at each DIT level for JBoss

Version	DIT=1	DIT=2	DIT=3	DIT=4	DIT=5	Total
1	419	165	240	477	14	1315
2	514	204	249	536	24	1527
3	514	204	249	536	24	1527
4	535	213	99	696	25	1568
5	590	62	68	0	0	720
6	590	62	68	0	0	720
7	590	62	68	0	0	720
8	590	62	68	0	0	720
9	590	62	68	0	0	720
10	590	62	68	0	0	720
11	590	62	68	0	0	720
12	590	62	68	0	0	720
13	590	62	68	0	0	720
14	590	62	68	0	0	720
15	590	62	68	0	0	720
16	590	62	68	0	0	720
17	590	62	68	0	0	720
18	590	62	68	0	0	720
19	590	62	68	0	0	720
20	590	62	68	0	0	720
21	590	62	68	0	0	720
22	590	62	68	0	0	720
23	590	62	68	0	0	720
24	590	62	68	0	0	720
25	591	62	68	0	0	721
26	591	60	68	0	0	719
27	612	71	69	0	0	752
28	636	93	134	0	0	863
29	638	96	134	0	0	868
30	627	110	146	0	0	883
31	627	110	145	0	0	882
32	669	111	140	0	0	920
33	677	109	140	0	0	926
34	676	102	139	0	0	917
35	679	103	137	0	0	919
36	725	102	149	0	0	976
37	803	108	148	0	0	1059
38	810	110	151	0	0	1071
39	822	113	154	0	0	1089
40	922	113	172	0	0	1207
41	921	113	179	0	0	1213
42	926	114	179	0	0	1219
43	921	114	179	0	0	1214
44	920	108	173	0	0	1201
45	931	108	173	0	0	1212

Table D.8. Number of attributes at each DIT level for Tyrant

References

Abreu, F., 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 8 pages.

Advani, D., Hassoun, Y., 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 ACM Symposium on Applied Computing (SAC 2006).* Dijon, France, pp.1713-1720.

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

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

Arisholm, E., Briand, L.C., Fuglerud, M.J. (2007) Data Mining Techniques for Building Fault-Proneness Models in Telecom Java Software. *Simula Technical Report.* 01-2007.

Basili, V. R., (1990) Viewing maintenance as reuse-oriented software development. *IEEE Software,* 7(1):19-25.

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

Basili, V.R., Briand, L.C., Melo, W.L. (1996b) How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104-116.

Belady, L.A. and Lehman, M.M. (1972) Introduction to Growth Dynamics. *Proceedings of Conference on Statistical Computing, Performance Evaluation,* Brown University. 1971, Academic Press, WFreiberger (ED), pp.503-511.

Bergel, A., Ducasse, S., Nierstrasz, O. (2005) Classbox/j: Controlling the Scope of Change in Java. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. California, USA, pp.177-189.

Bieman, J.M. and Zhao, J. (1995) Reuse through inheritance: A quantitative study of C++ software. *Proceedings of the 1995 ACM Symposium on Software Reusability.* Seattle, Washington, USA, pp.47-52.

Booch, G. (1993) *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, California, USA.

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

Briand, L.C., Daly, J., Wust, J. (1998) A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering-An International Journal,* 3(1):65-117.

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

Briand, L.C., Melo, W.L. and Wust, J. (2002) Assessing the applicability of fault-proneness models across OO software projects. *IEEE Transactions on Software Engineering,* 28(7):706-720.

Briand, L.C. and Wust, J. (2002) *Empirical Studies of Quality Models in Object-Oriented Systems*. Advances in Computers, Vol. 59. Academic Press, pp.97-166.

Briand, L.C., Wüst, J., Daly, J.W. and Victor Porter, D. (2000) Exploring the relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51(3):245-273.

Briand, L.C., Wust, J.K. and Lounis, H. (2001) Replicated Case Studies for Investigating Quality Factors in OO Designs. *Empirical Software Engineering*, 6(1):11-58.

Briand, L.C., Wust, J., Lounis, H., and Ikonomovski, S.V. (1999c) Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study. *Proceedings of 21st International Conference on Software Engineering*. Los Angeles, USA. pp.345-354.

Buckley, J., Exton, C., Good, J. (2004) Characterizing programmers' information-seeking during software evolution. *Proceedings of the 12th International Workshop on Software Technology and Engineering Practice*. Illinois, USA. pp.23-29.

Buckley, J., Mens, T., Zenger, M., Rashid, A. and Kniesel, G. (2005) Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice,* 17(5):309-332.

Capiluppi, A., Morisio, M., and Ramil, J. (2004) Structural Evolution of an Open Source System: A Case Study. *Proceedings of the 12th Internaltional Workshop on Program Comprehension.* Bari, Italy, pp.172-182.

Capiluppi, A. and Ramil, J. (2004) Studying the evolution of open source systems at different levels of granularity: Two case studies. *Proceedings of 7th International Workshop on Principals of Software Evolution.* Kyoto, Japan, pp.113-118.

Cartwright, M. (1998) An empirical view of inheritance. *Journal Information and Software Technology*, 40(14):795-799.

Cartwright, M., and Shepperd, M. (2000) An Empirical Investigation of an object-oriented (OO) system. *IEEE Transactions on Software Engineering*, 26(8):786-796.

Chidamber, S.R., Darcy, D.P., Kemerer, C.F. (1998) Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629-639.

Chidamber, S.R., Kemerer, C.F. (1994) A metrics suite for object-oriented design. *IEEE Transaction on Software Engineering*, 26(8):476-493.

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

Cohen, J., Cohen, P., West, S.G. and Aiken, L.S. (2003) *Applied Multiple Regression/Correlation Analysis for the Behavioral Science* (3rd edition). Hillside, NJ: Lawrence Erlbaum Associates.

Counsell, S. (2008) An Analysis of Faulty and Fault-Free C++ Classes Using an Object-Oriented Metrics Suite. *Book Chapter: Innovative Techniques in Instruction Technology, E-learning, E-Assessment, and Education*. pp.520-525.

Counsell, S., Hassoun, Y., Johnson, R., Mannock, K., Mendes, E. (2003) Trends in Java Code Changes: the key to identification of refactorings. *Proceeding of the 2nd International Conference on the Principles and Practice of Programming in Java.* Kilkenny, Ireland, pp.45-48.

Counsell, S., Hassoun, Y., Loizou, G., Najjar, R. (2006a) Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06).* Rio de Janeiro, Brazil, 288-296.

Counsell S, Loizou G, Najjar. R (2006b) Ignore size and inner classes, poor class layout and feature order are the real enemies of OSS developers. *Birkbeck Technical Report*, BBKCS-06-08.

Counsell, S. and Swift, S. (2008) An Empirical Study of Potential Vulnerability Faults in Java Open-Source Software. *Book Chapter: Innovative Techniques in Instruction Technology, E-learning, E-Assessment, and Education.* pp.514-519.

Daly, J., Brooks, A, Miller, J., Roper, M. and Wood, M (1996) Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. *Empirical Software Engineering: an International Journal*, 1(2):109-132.

Deligiannis, I., Shepperd, M., Roumeliotis, M., Stamelo, I. (2003) An empirical investigation of an object-oriented design heuristic for maintainability. *The Journal of Systems and Software*. 65(2):127-139.

DeMarco, T. (1982) *Controlling Software Projects*. Yourdon Press, New York, NY.

Demeyer, S., Ducasse, S., Nierstrasz, O. (2000) Finding refactorings via change metrics. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Minneapolis, MN, pp.166-177.

Dvorak, J. (1994) Conceptual Entropy and its Effct on Class Hierarchies. *Computer*, 27(6):59-63.

El Emam, K., Benlarbi, S., Goel, N., Rai, S. (2001) The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):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., Pfleeger, S.L. (2002) *Software Metrics: ARigorous and Practical Approach*. International Thomson Computer Press, London, UK.

Fenton, N., Pfleeger, S. L. And Glass, R. (1994) Science and Substance: A Challange to Software Engineers. *IEEE Software*, 11 (4):86-95.

Field, A. (2006) *Discovering Statistics Using SPSS.* Sage Publication, London, UK.

FindBugs Tool (2008) http://findbugs.sourceforge.net/

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

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

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

Girba, T.; Lanza, M.; Ducasse, S. (2005) Characterizing the Evolution of Class Hierarchies. *Proceedings of the 9th European International Conference on Software Maintenance and Reengineering.* Manchester, UK, pp.2-11.

Glasberg, D., El Emam, K., Melo, W., Madhavji, N. (2000) Validating OO Design Metrics on a Commercial Java Application. *Nat. Res. Council Canada, Institute for IT* (NRC/ERB-1080).

Hall, T., Rainer, A., 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., Nithi, R. (1998a) Coupling metrics for OO design, *Proceedings of the 5th International Software Metrics Symposium (METRICS 1998)*. Bethesda, MD, pp.150-157.

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

Harrison, R., Counsell, S., Nithi, R. (2000) Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3):173-179.

Henderson-Sellers, B. (1995), *Object-oriented metrics: measures of complexity*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

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

Hinkle, D. E., Wiersma, W. and Jurs, S., G. (1995) *Applied Statistics for the Behavioural Science*, Boston: Houghton Mifflin.

JHawk Tool (2008): Available at: http://www.virtualmachinery.com/jhawkprod.htm .

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

Kemerer, C.F. and Slaughter, S. (1999) An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, 25(4):493-509.

Kerievsky, J. (2004) *Refactoring to Patterns*. Addison Wesley, Reading, MA. Also partially available online at: www.indstriallogic.com, 2002.

Kitchenham, B. (2004) *Procedures for Performing Systematic Reviews*, Keele University.

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

Kitchenham, B.A., Pfleeger, S.L., Pickard, L,. Jones, P., Hoaglin, D., El Emam, L., and Rosenberg, J. (2002) Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721-734.

Lehman, M.M. (1974) Programs, Cities, Students, Limits to Growth? Inaugural Lecture, May 1974, *published in Imperial College of Science and Technology. Inaugural Lecture Series*, vol. 9, pp. 211-229. 1970-1074. *Also in Programming Methodology, D. Gries Ed* (1979). New York: Springer-Verlag, pp.42-69.

Lehman, M.M. (1978) Laws of Program Evolution – Rules and Tools for Programming Management. *Proceedings of Infotech State of the Art Conference,* Why Software Projects Fail, pp.11/1-11/25.

Lehman, M.M. (1980a) Understanding Laws, Evolution and Conservation in the Large Program Life Cycle. *The Journal of Systems and Software*, 1(3):213-221.

Lehman, M.M. (1980b) Programs, life Cycle, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):060-1076.

Lehman, M.M. (1996) Laws of software evolution revisited. Position Paper, *5th European Workshop on Software Process Technology,* Trondheim, Norway, LNCS 1149, Springer Verlag, pp.108-124.

Lehman, M.M. and Ramil, J.F (2001) Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering,* 11(1):15-44.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., and Turski, W.M. (1997) Metrics and Laws of Software Evolution - The Nineties View. *Proceedings of the* Fourth *International Symposium on Software Metrics (METRICS 97).* pp.20-32.

Lewis, J., Henry, S., Kafura, D., Schulman, R. (1991) An Empirical study of the object-oriented paradigm and software reuse. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Phoenix, AZ, pp.184-196.

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

Lorenz, M., Kidd, I. (1994*) Object-Oriented Software Engineering Metrics*, Prentics-Hall Englwood Cliff, NJ.

McCabe, T. (1976) A software complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308-320.

McGraw, G. and Felten, E. (1998) Twelve rules for developing more secure Java. *Java World*, 12/01/1998. available at: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html?page=1.

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

Meyers, W. (1988) Interview with Wilma Osborne. *IEEE Software*, 5(3):104-105.

Najjar, R., Counsell, S., Loizou, G. and Mannock, K. (2003) The Role of Constructors in the Context of Refactoring Object-Oriented Systems. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering.* Benevento, Italy. pp.111-120.

Nasseri, E. and Counsell, S. (2008) Inheritance, ‘Warnings’ and potential Refactorings. *Proceedings of the 3rd International Conference on Software Engineering Advances*. Sliema, Malta. pp.132-39.

Nasseri, E., and Counsell, S. (2009a) System evolution at the attribute level: an empirical study of three Java OSS and their refactorings. P*roceedings of the 31st International Conference on Information Technology Interface*, Cavtat, Dubrovnik, Croatia. pp.653-658.

Nasseri, E. and Counsell, S. (2009b) An Empirical study of Java System Evolution at the Method Level. *Accepted to appear in the Proceedings of the 7th International Conference on Software Engineering Research Management and Applications*. Haikou, Hainan Island, China.

Nasseri, E., and Counsell, S. (2009c) An Evolutionary Study of Inheritance and Method Invocation in Java OSS. *Accepted (subject to minor changes) to appear in the Software Quality Journal*.

Nasseri, E., Counsell, S. and Shepperd, M. (2008) An Empirical Study of Evolution of inheritance in Java OSS. *Proceedings of the 19th Australian Software Engineering*. Conference, Perth, Australia. pp.269-278.

Nasseri, E., Counsell, S. and Shepperd, M. (2009) Class Movement and Re-location: an Empirical Study of Java Inheritance Evolution. *Accepted (subject to minor changes) to appear in The Journal of Systems and Software*.

O'Brien, M.P., Buckley, J., Exton, C. (2005) Empirically studying software practitioners - bridging the gap between theory and practice. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM ’05),* Budapest, Hungary. pp. 433-442.

Opdyke, W. “Refactoring object-oriented frameworks.” Ph.D. Thesis, University of Illinois, 1992.

Opdyke, W.F., Johnson, R.E. (1993) Creating abstract superclasses by refactoring. *Proceedings of the ACM 1993 Computer Science Conference.* Indianapolis, IN, pp. 66-73.

Opdyke, W.F., Johnson, R.E. (1990) Refactoring: and 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.

Ostrand, T.J., Weyuker, E.J., Bell, R.M. (2004) Where the Bugs are. *Proceedings of the 2004 ACM SIGSOFT Internaltional Symposium on Software Testing and analysis.* pp.86-96.

Perry, D.E., Porter, A.A. & Votta, L.G. (2000) Empirical studies of software engineering: a roadmap. *Proceedings of the conference on The future of Software engineering.* Limerick, Ireland , pp.345-355.

Ping, Y., Systa, T., Muller, H. (2002) Predicting fault-proneness using OO metrics. An industrial case study. *Proceedings of the 6th European Conference on Software Maintenance and Reengineering.* Budapest, Hungary, pp.99-107.

Prechelt, L., Unger, B., Philippsen, M., and Tichy, W. (2003) A controlled experiment on inheritance depth as a cost factor for code maintenance. *The Journal of Systems and Software,* 65(2):115-126.

Rosenberg, J. (1997) Some misconceptions about lines of code. *Proceedings of the 4th International Software Metrics Symposium*. Albuquerque, NM, pp.137-142.

Rumbaugh, J., Jacobson, I., Booch, G. (1998) *The Unified Modelling Language Reference Manual.* Addison Wesley, Reading, MA.

Sangwan, R.S., Vercellone-Smith, P. and Laplante, P.A. (2008) Structural Epochs in the Complexity of Software over Time. *IEEE Software,* 25(4):66-73.

Seaman, C. (1999) Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557-572.

Shepperd, M.J. (1995) *Foundations of Software Measurement*. Prentice Hall International, Hertfordshire, UK.

Sintes, T. (2001) So what are inner classes good for anyway? *Java World*, 27/07/01. available at: http://www.javaworld.com/javaworld/javaqa/2000-03/02-qa-innerclass.html?page=1.

Skogland, M. (2003) Practical use of encapsulation in object-oriented programming. *Proceedings of the 2003 International Conference on Software Engineering Research and Practice*. Las Vegas, NV, pp.554-560.

Snyder, N. (1986) Encapsulation and inheritance in object-oriented programming Language. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, pp.38-45.

Stamelos, I., Angelis, L., Oikonomou, A. and Bleris, L. (2002) Code quality analysis in open source software development. *Information Systems Journal,* 12(1):43-60.

Swanson, E.B. (1976) The Dimensions of Maintenance. *Proceedings of the 2nd IEEE International Conference On Software Engineering.* San Francisco, California, pp.492-497.

Tempero, E.D., Noble, J., Melton, H. (2008) how Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. *Proceedings of the 22nd European Conference on Object-Oriented Programming*. Paphos, Cyprus, pp.667-691.

Tilevich, E., and Smaragdakis, Y. (2005) Binary refactoring: improving code behind the scenes. *Proceedings of the 27th International Conference on Software Engineering (ICSE).* St Louis, USA, pp.264-273.

Tokuda, L., Batory, D. (2001) Evolving object-oriented designs with refactorings. *Journal of Automated Software Engineering,* 8(1):89-120.

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.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6): pp.80-83.

Wood, M., Daly, J., Miller, J. and Roper, M. (1999) Multi-method research: An empirical investigation of object-oriented technology. *The Journal of Systems and Software,* 48(1):13-26.