An Empirical Investigation into Contributory Factors of Change and Fault Propensity in Large-Scale, Commercial, Object-Oriented Software

By

Matt Gatrell

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy

March 2012

Department of Information Systems and Computing

Brunel University

ABSTRACT

Object-Oriented design and development dominates both commercial and open source software projects. One of the principal goals of object-oriented design is to aid reuse, and hence, reduce future maintenance efforts of software systems. However, the on-going maintenance of large-scale software systems (both changes and faults) continues to be a significant proportion of the lifecycle of the system and the total investment cost. Understanding and thus being able to predict - or even reduce - the impact of the contributing factors of future maintenance efforts of a software system is thus highly beneficial to software practitioners. In this Thesis we empirically study a large, commercial software system with the principal aim to determine the contributing factors to the change and fault propensity over a three-year period. We consider the object-oriented design context of the software, specifically its inheritance characteristics, coupling and cohesion properties, object-oriented design pattern participation, and size. We also explore the effect of refactoring and test classes in the software. Our results show that several aspects of the design context of a class have an impact to the change and fault-proneness of the software. Specifically, we show that classes with high afferent or efferent coupling are more change and fault-prone; we also identify a number of design patterns whose participants tend to have a higher change and fault propensity than non-participants and we identify a range of inheritance characteristics (in terms of depth of inheritance and number of children) that result in an increase to change and fault-proneness. Furthermore we show that refactoring is a commonly occurring maintenance activity, although it is largely limited to simpler types of refactorings. Finally, we provide some insight into the co-evolution of production and test code during refactoring.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Dr. Steve Counsell, firstly, as the source of inspiration to embark upon the research several years ago when I was a Masters student in one of Steve's classes, and secondly, for Steve's unbounded support, guidance, patience and encouragement. Above all, Steve has made the experience a rewarding and enjoyable one and my greatest thanks are for that.

I would also like to thank my patient wife, Anna, who continues to provide endless support and encouragement to this and every other endeavour. And it is to Anna that this Thesis is dedicated with thanks and love.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

LIST OF PUBLICATIONS

This section lists the publications that form the basis of the research in this thesis.

2009

Gatrell, M., & Counsell, S. (2009). Empirical Support for Two Refactoring Studies Using Commercial C# Software. *13th International Conference on Evaluation & Assessment in Software Engineering.* Durham.

Gatrell, M., Counsell, S., & Hall, T. (2009). Design patterns and change-proneness: a replication using proprietary C# software. *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'09),* pp160-164. Lille.

2010

Gatrell, M., & Counsell, S. (2010). Size, Inheritance, Change and Fault-proneness in C# software. *The Journal of Object Technology , 9* (5), pp29-54.

2011

Gatrell, M., & Counsell, S. (2011). Design Patterns and Fault-Proneness: A Study of Commercial C# Software. *Proceedings of Fifth IEEE International Conference on Research Challenges in Information Science (RCIS 2011).* Guadeloupe. Selected as one of the best papers of the conference and published in an extended journal.

2012

Gatrell, M., & Counsell, S. (2012). Faults and their Relationship to Implemented Patterns, Coupling and Cohesion in Commercial C# Software. *(To appear in) International Journal of Information System Modeling and Design*.

1 CHAPTER 1 – INTRODUCTION

1.1 INTRODUCTION

In this chapter we introduce the background to our research and summarise the core aim of the Thesis. We then provide a more detailed analysis of the core subject areas before describing how our study relates to these topics. We discuss the motivation behind our research and the detailed objectives of the study. Finally, we provide an overview of the structure of the Thesis.

1.2 BACKGROUND

Object-Oriented software development has become a common paradigm for software development, with the popularity of Java, C++ and C# object-oriented software programming languages dominating both commercial and open source software projects. One of the principal goals of object-oriented design is to aid reuse, and hence, reduce future maintenance efforts of software systems. Whether this has proven to be true, however, is an open research question. The on-going maintenance of large-scale software systems (as a result of both changes and faults) continues to comprise a significant proportion of the lifecycle of the system and the total investment cost. Understanding and thus being able to predict - or even reduce - the impact of the contributing factors to future maintenance efforts of a software system is thus highly beneficial to software practitioners.

The foundation of this Thesis lies with the uninhibited access to the version control and fault tracking systems for a large-scale commercial software system over a three-year period. We set out to study the change behaviour of the software system; more specifically, the aim of this Thesis is to determine contributory factors to change and fault-proneness. We consider the design context of the software and maintenance activities and measure the effect to the change and fault propensity of the software over that three-year period. We consider several aspects of the design context of the software in relation to the change and fault propensity: inheritance characteristics, coupling and cohesion properties, size, and design pattern participation. We also explore the impact of refactoring and test code.

The software system and associated data studied featured several factors of interest that influenced the direction of our study. First, the change and fault data available was detailed and granular. From the beginning of the study, a check-in policy was enforced on the version control system meaning that any change committed to the version control system had to be manually attributed to a change or a fault in the fault tracking system by the developer making the change. No modification could be made to the software system without being tracked against a recorded change or fault. Second, the system design is object-oriented and is heavily influenced by commonly documented object-oriented design patterns. Third, refactoring (a maintenance process to improve the internal characteristics of software, while preserving its external behaviour) was thought to be a commonplace activity during the development of the system. Fourth, the software is developed using a test-driven methodology and, as such, a significant proportion of the source code consists of test classes. Finally, the software system is a commercially developed and deployed software system and thus offers an interesting insight into commercially developed software. It also offers comparisons with studies of open source software.

1.3 AN INTRODUCTION TO THE CORE SUBJECT AREAS

In this section we provide an overview of the main subject areas relating to our research.

1.3.1 OBJECT-ORIENTED DESIGN

This Thesis presents a study of change and fault-proneness in object-oriented software. There are several facets to the object-oriented paradigm and in this section we explore these to provide a backdrop to the study. We highlight the specific topic areas that relate to the empirical experiments that form part of this Thesis. The principal concepts of the object-oriented paradigm are *encapsulation*, *polymorphism* and *inheritance*.

Encapsulation restricts access to the internal workings of a software module, or class, so that only its public data or methods are exposed to an externally calling client. This mechanism can prohibit the setting of internal data, or calling of internal methods, by marking them as *private*, ultimately controlling interaction with a class to prevent it being left in an inconsistent state. Encapsulation can prevent unauthorised access and

is said to increase robustness as a consequence. It also reduces complexity (by, for example, improving the comprehension) of a class by limiting the interdependencies (coupling) between classes.

Polymorphism is the ability of a single object to have several forms. There are several types of polymorphism: the most commonly known is subtype polymorphism where a single class can implement multiple interfaces, allowing the same object to appear as a different type depending on the interface through which it is accessed. Other types of polymorphism include operator overloading, where an operator (such as '+' or '-') can have different behaviour depending on the type of the operands. Method overloading is a form of polymorphism that allows the same method to behave differently depending on the number and type of parameters passed into it. Finally, parametric polymorphism allows a class to be defined that will behave differently based on which type it is constructed with. In C# and Java, generics are used to create parametric polymorphic types, whereas in C++, templates are used.

Inheritance aids reuse by allowing a class to inherit functionality from a parent or base class. The inherited behaviour can then be extended or overridden at the child class level. Typically, only single inheritance (that is, a class can have only one direct parent class) is supported (Java, C#) but some languages (such as C++) allow multiple inheritance. Inheritance is said to aid program understanding, however there is conflicting evidence of its fit for purpose. Several studies (Daly, 1996), (Snyder, 1986) have suggested that, rather than aid understanding, the opposite is true. There are several specific documented examples of this, such as the *fragile base class* example (Mikhajlov & Sekerinski, 1999). A *fragile base class* exists when a change to a base class causes unpredictable behaviour to ripple through sub classes in the inheritance hierarchy; in many cases this can render the base class too sensitive to reliably change. As such, inheritance is often regarded as the most contentious of the three concepts that constitute object-oriented design. Specifically the use of deep inheritance hierarchies or large number of sub classes, has been cited (Wood, 1999), (Prechelt L. U., 2003), (Cartwright M. a., 2000) as a contributing factor to change and fault-proneness in software and as such, is of particular interest to our study and is the subject of an empirical investigation documented in Chapter 4 of this Thesis.

1.3.2 COUPLING

Coupling measures the inter-dependencies between classes, usually the sum of the number of dependants and dependencies of a class (Briand, Daly, & Wust, 1999). Inheritance is considered a particularly tight form of coupling due to the close bond between a base class and its children (a change in a base class can directly and dramatically effect the behaviour of a child class). There are several other forms of coupling, such as composition where are class is referenced, usually through a strictly defined interface, and used within the execution of the classes method, and is considered more loosely coupled than inheritance. Using composition, it can be far easier to replace the class with an alternative offering a different implementation, when compared to changing the behaviour of an inherited class in a large inheritance hierarchy. Classes with lower, or looser, coupling are preferred as they tend to be easier to reuse and are less prone to change resulting from changes rippling through a dependancy chain. As such, a class with low coupling may be expected to be less fault-prone than a class with high coupling.

In this Thesis we use two distinct forms of coupling to inform our empirical studies. *Afferent* coupling measures the number of types that directly depend on it. *Efferent* coupling measures the number of types that a type directly depends on (Li & Henry, 1993).

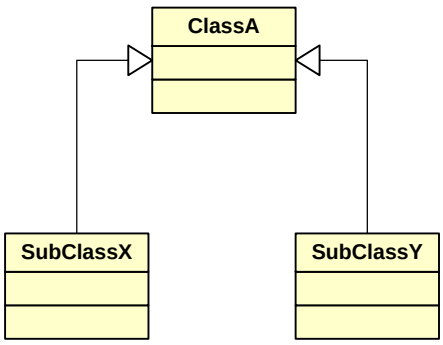

FIGURE 1-1 EFFERENT AND AFFERENT COUPLING

Figure 1-1 shows a simple class structure: **SubClassX** and **SubClassY** both inherit **ClassA**; in this example **Class A** has an afferent coupling of 2, as **SubClassX** and **SubClassY** directly depend on it; **SubClassX** and **SubClassY** have an efferent coupling

of 1, as they both directly depend on **ClassA**. The close association between coupling and many aspects of object-oriented software development (such as inheritance, design patterns and refactoring) results in coupling commonly being considered in studies of object-oriented software (Basili, Briand, & Melo, 1996) (Briand, Daly, & Wust, 1999); equally, coupling forms a key part of our study and is included in the empirical investigation documented in Chapter 4.

1.3.3 COHESION

Cohesion measures how strongly related the functionality within a class is (Briand, Daly, & Wust, 1997). Classes with relatively higher cohesion tend to be more reusable and comprehensible as they are more likely to be a self contained module. As such, a cohesive class may be expected to be less fault-prone than an uncohesive class. Cohesiveness could be considered a fairly subjective term (Counsell, Swift, & Mendes, 2002). Many attempts have been made to measure cohesiveness (Chidamber & Kemerer, 1994) the most common of which uses a measurement of how interconnected the fields and methods from within a single class are (that is, if all methods in a class access all fields of the same class, then the class is considered cohesive; if all methods in a class do not access any of the fields in the class, then the class is considered to be uncohesive).

The coupling and cohesion of a class are often considered together as a pair of measurements and are closely connected with the core concepts of object-oriented design. As such, they are of particular interest to us when studying the effect of the design context of a class on its change and fault propensity and we consider cohesion as part of the empirical investigation documented in Chapter 4.

1.3.4 DESIGN PATTERNS

Object-Oriented design patterns are reusable descriptions or templates showing the relationships and interactions (coupling) between classes and objects. Many design patterns including some of those described by Gamma et al., (Gamma, 1995), promote adaptability by supporting specialization (inheritance) of the pattern-based classes. A system built using design patterns can therefore be adapted by creating concrete classes with desired new functionality rather than by direct modification of the existing set of

core pattern-based classes. It is therefore reasonable to expect that design pattern 'participant' classes (i.e., those core classes) would have a relatively lower propensity for change relative to all other classes in a system because, in theory, they should remain untouched by developers. There are, however, studies that show the contrary (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003); because of Bieman's somewhat surprising result, we include design patterns in our study into change and fault-proneness, documenting an empirical investigation into design pattern participation and change and fault-proneness in Chapter 5.

An example of a design pattern is the *adaptor* design pattern, illustrated in Figure 1-2. The *adaptor* design pattern allows classes to communicate that would not normally work together because of an incompatible interface. The adaptor converts the interface of a class (in Figure 1-2 below the **Adaptee***)* into another interface that the client expects (in Figure 1-2 below the **Target**) allowing the **Client** to call the **Adaptee** as if it had the **Target** interface.

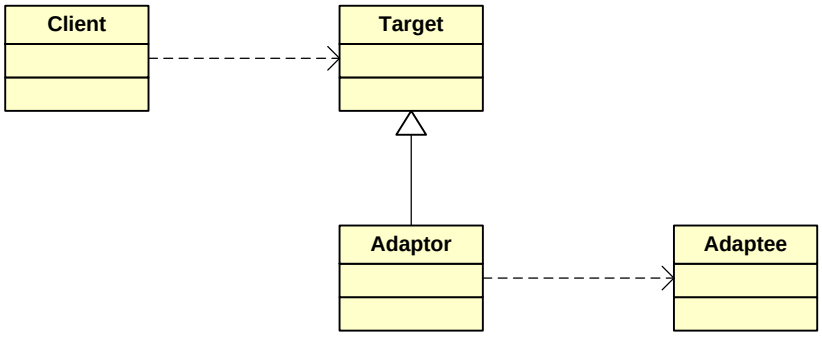

FIGURE 1-2 THE ADAPTOR DESIGN PATTERN

1.3.5 REFACTORING

Refactoring is the process of applying one or more (often small) changes to source code to improve the internal structure of the software, while preserving the external behaviour. There are many types of refactoring ranging from improving the comprehension of a class, to improving the coupling or cohesiveness of a class by, for example, moving methods or fields up or down an inheritance hierarchy. The motivation that lies behind refactoring is with its perceived ability to slow down, or

even reverse, code decay. Slowing or reversing code decay is a desirable outcome as the resulting software becomes easier to comprehend and requires less effort to maintain in the longer term. Refactoring has become increasingly common in recent years partly as a result of greater industry acceptance and tool support (Mens & Tourwe, 2004). There are, however, opposing views of the benefits of refactoring. First, there is little quantitative evidence to show that the benefits of refactoring outweigh the additional effort spent in the refactoring process in commerical software. Furthermore, it is widely accepted that one of the biggest contributing factors to fault-proneness in software is change; that is, if a class is changed then it is more likely to contain a fault. Considering this, there is little quantitative evidence to show that the benefits of refactoring are outweighed by the additional effort required to fix faults introduced as part of the refactoring process in commercial software. A core part of this Thesis explores refactoring activities and we show profile refactoring activities in an empirical investigation in Chapter 3. An example of a refactoring is *pull-up field* illustrated in Figure 1-3. In a *pull-up field* refactoring two classes exist in the same inheritance hierarchy and have the same field. The refactoring results in the field being pulled up to the super class and removed from the two original classes which now inherit the field from their parent class. Now the field only exists in one place there is a reduced chance of duplicate code required to access, set or validate the field. In Figure 1-3 the classes **Salesman** and **Engineer** inherit the class **Employee**. In their initial state, before the pull-up field refactoring, the field name is defined in both **Salesman** and **Engineer** classes. After the refactoring has been applied, the field name is moved to the super class, **Employee**, and is removed from the **Salesman** and **Engineer** classes.

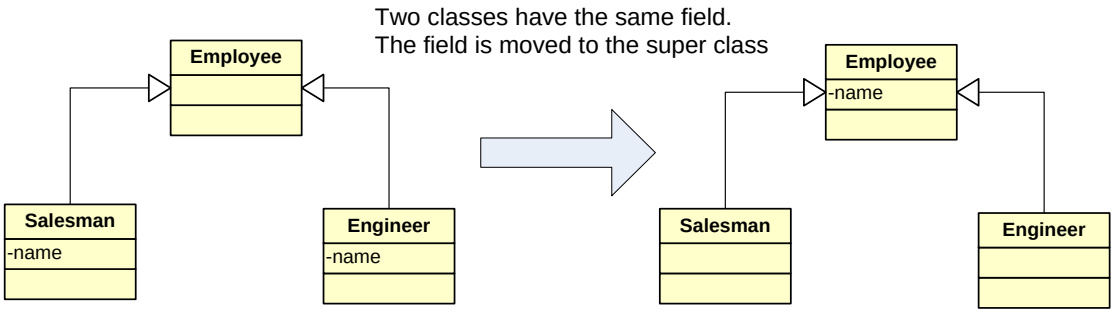

FIGURE 1-3 PULL-UP FIELD REFACTORING

1.3.6 UNIT TESTING

Automated unit tests, using the xUnit framework (for example JUnit (JUnit.org) for Java or NUnit (NUnit.org) for C#), are commonly used when refactoring, as they offer a way to quickly guarantee the external behaviour of the class is consistent before and after a refactoring has taken place. The xUnit framework allows a developer to write repeatable test cases against a unit of code in the same programming language as the implementation code. Test cases can test both the normal execution path through the unit of production code and exception paths and, as such, the amount of test code that must be maintained can be significant. When the production code is evolved, either through enhancement or fault fixing, the test code is likely to also to be evolved; this is called co-evolution. Due to the close association of test code with refactoring, we consider the impact of test code during our empirical investigations into refactoring in Chapters 3 and **Error! Reference source not found.**.

1.4 THE STUDY CONTEXT

The aim of this Thesis is to determine the contributing factors to the change and fault propensity of the studied system. We consider the design context of the software, specifically the inheritance characteristics, coupling and cohesion properties, size, and design pattern participation. We also explore the effect that refactoring and test code has on the change and fault-proneness of the software.

Figure 1.4 shows the time line of the system that we used for the set of empirical studies described in the Thesis. We provide an overview of this timeline at this point, with more detail accompanying each study in the Thesis.

The anonymous system used as a basis of the empirical study is from a large, international software company specialising in transaction content processing software. The system relates to a core technology product written in C#, Java and C++. A subset of the system, written in C# by a team of 8-10 developers, is the focus of the study; this subset was chosen due to the more detailed change and fault data available for this part of the system, when compared to the Java and C++ parts of the system. The system had been running for approximately 24 months from when our analysis started and was under active development throughout the study; it includes server side components, a web application and a number of client side components and tools. Henceforward, for

confidentiality purposes, we will refer to the system as 'WebCSC', the studied subset comprising over 7400 classes (at the latest studied version). The WebCSC system was developed in a continuous integration environment where every change committed to source control by a developer was explicitly versioned and built by the continuous integration server; this allowed us to compare code at each version step. On average, a version was submitted to the source control system every hour over the period studied. A significant portion of the source code comprised (unit) test classes and we include these as well as production classes as part of our analysis.

At the early stages of the software systems lifecycle we recorded the type and frequency of refactoring that occurred. The data was collected from early 2006. In the two year period (2006 and 2007) we undertook several studies to explore the link between the change and fault-proneness and any of the key aspects of the design context of the software: design pattern participation, coupling and cohesion properties, inheritance characteristics and size. During the refactoring study we also investigated to what extent the test code was refactored when compared to the production code.

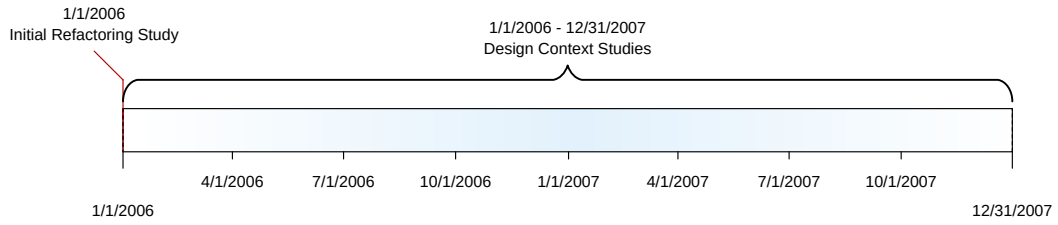

FIGURE 1-4 TIMELINE OF STUDY

1.5 MOTIVATION

The motivation for our study stems from several sources. Principally, the ongoing maintenance costs of large-scale software systems continue to be considerable, despite the predictions of a reduction in maintenance costs as a result of the additional reuse and comprehension offered by the object-oriented paradigm several decades ago. What causes software to be more fault and change-prone when compared with other engineering disciplines? Many of the reasons lie in areas independent of the build (design and development) of the software: requirement and market analysis, rapid change of underlying technologies and emerging standards. However, some blame must

lie within the design and development of the software itself: design choices, programming standards, and adopted methodologies.

To identify the contributing factors to change and fault-proneness gives us (software practitioners) the opportunity to predict (more accurately) the future maintenance efforts required for a software system. Better still, we may be able to reduce the impact of the contributing factors resulting in less change or fault-prone software. Of particular benefit is the ability to identify influencing factors of change and fault-prone early in the software development lifecycle that can be mitigated or eliminated, perhaps even before the software is built. For example, if there is a particular aspect of the design context that can be detected and avoided at design time, then we can implement a set of refactorings that should be targeted or avoided, the use of a specific design pattern or a testing methodology. Additionally, there are several emerging (or emerged) software engineering practices that have achieved (at least partial) industry adoption and tool support despite little knowledge or quantitative evidence of their benefit; refactoring and test-driven development are two examples. If we can identify the effect of refactoring and test-driven development on the change behaviour of the commercial software system studied, we will be able to conclude whether these practices are contributory factors to change and fault-proneness, either positively or negatively. Finally, the studied system is a large-scale commercial software system under active development, the opportunity to study the system over a prolonged period offers the opportunity to make comparisons to other open source studies to see if trends are common across both paradigms. Finally, to see if the trends change as a system matures is still an open research issue..

1.6 AIMS, OBJECTIVES AND CONTRIBUTIONS

The aim of this Thesis is to identify contributory factors to change and fault-proneness in the studied software system; in order to achieve this, the corresponding objectives and contributions of this Thesis are:

To understand how refactorings are applied in commercial software. Are refactorings regularly applied, and if so what types of refactorings are most common? Are refactorings inter-dependent (that is, if refactoring x is applied is there a likelihood that

refactoring y will also be applied?). Does the application of refactorings in commercial software differ from what we have previously seen in open source studies?

Second, to understand how design patterns are applied in commercial software. Which patterns are most common? How often are they applied? Does design pattern participation result in less change-prone and fault-prone classes?

Finally, are the key aspects of the object-oriented design context: inheritance, coupling and cohesion, and size, contributory factors of fault and change propensity? Are there properties or characteristics of these aspects of the design context that cause fault or change-proneness more than others?

1.7 RESEARCH FRAMEWORK

Empirical software engineering is a sub-domain of software engineering that focuses on devising experiments on software, collecting data from the experiments, identifying trends in the collected data to lead to theories and laws and to accumulate knowledge (Kitchenham, et al., 2008). In this Thesis we conduct a series of empirical investigations in to the studied commercial software system. To collect the data for the empirical investigations we developed a bespoke tool that mines the software for key attributes and metrics; we also use the metrics, data and associated built-in tools available as part of the source control system and the fault tracking system. When analysing the data we use a variety of statistical analysis techniques, ranging from graphical depictions of the data (such as box plots) to correlations to identify trends and draw conclusions. Parametric (Pearson) and non-parametric (Spearman) correlations are used throughout our study. There are some considerations which must be taken into account, when evaluating correlations: correlations show an association between two variables but do not necessarily imply causation. For example, there may be a third - or many other - hidden variables, that are the underlying cause of the observed association. Finally, correlations over large sample sizes – such as the samples in our studies – can become erratic and produce worse results than correlations of smaller samples. Because of this, we also apply non-parametric statistical tests where appropriate.

1.8 OVERVIEW OF THE THESIS

Chapter 2 describes work related and complimentary to the research areas contributed to in this thesis. We cover areas including refactoring, design patterns, inheritance characteristics, coupling and cohesion, size and testing. The chapter also adds insights into the motivation behind, and justifications for the direction of the thesis.

Chapter 3 presents an empirical study into refactoring in commercial software. We apply a bespoke source code analysis tool to 270 versions of the WebCSC. We examined each version with its predecessor to detect application of 15 types of refactorings in both production and test code classes. The results are compared to results from earlier studies on refactoring of Open-Source systems (Advani, 2006) and differences between test and production code (in terms of applied refactorings) investigated.

Chapter 4 documents an empirical study into the effect the design context of a class has on fault and change-proneness. Specifically, we explore change and fault propensity through inheritance characteristics, coupling and cohesion and size. Inheritance characteristics of each class were identified based on their inheritance depth and the number of subclasses (i.e., children) belonging to each. Coupling properties were calculated based on the number of dependents and dependencies a class had, and cohesion was measured using a common cohesion metric (the Henderson-Sellers version of the Lack of Cohesion in Methods metric, LCOM(HS)). The design context characteristics measured were compared to the change and fault history of the classes to determine the relationship.

Chapter 5 describes an empirical study into design patterns and their effect on change and fault propensity. We seek to determine whether design pattern participants are more change-prone in the studied system and we determine whether pattern classes are more fault-prone than non-pattern classes.

Chapter 6 presents the conclusions and contributions of our research and we associate them with the original objectives of the thesis. Future related research ideas are proposed and personal perspectives on completing this thesis discussed.

2 CHAPTER 2 – A SURVEY OF RELATED WORK

2.1 INTRODUCTION

In the last chapter we provided an introduction to the aims and objectives of the Thesis. In this chapter we extend this by discussing work related to the research areas of our study. We examine research in refactoring, testing, design patterns, inheritance, coupling and cohesion. Presenting the related research provides an insight into the subject areas that our research extends and to the current state-of-the-art in these areas.

2.2 REFACTORING

Refactoring is the process of improving the internal characteristics of existing software while preserving its external behaviour (Fowler M. , 1999). Bill Opdyke first documented the benefits of refactoring in his doctoral Thesis (Opdyke, 1992) and this was followed some years later by Martin Fowler's book *Refactoring: Improving the Design of Existing Code* (Fowler M. , 1999) where he documents 72 common refactorings, the manual process involved in applying each of them and gives examples of each.

Refactoring has been suggested as having a positive influence on the long-term quality and maintainability of software and to prevent design decay within software (Fowler M. , 1999). An example of a refactoring is *a push down method* refactoring. A push down method refactoring behaviour is applicable when one or more methods of a superclass is relevant only for some of its subclasses and so the method should be moved to those subclasses. Figure 2-1 shows the state of a class structure before and after a push down method refactoring is applied.

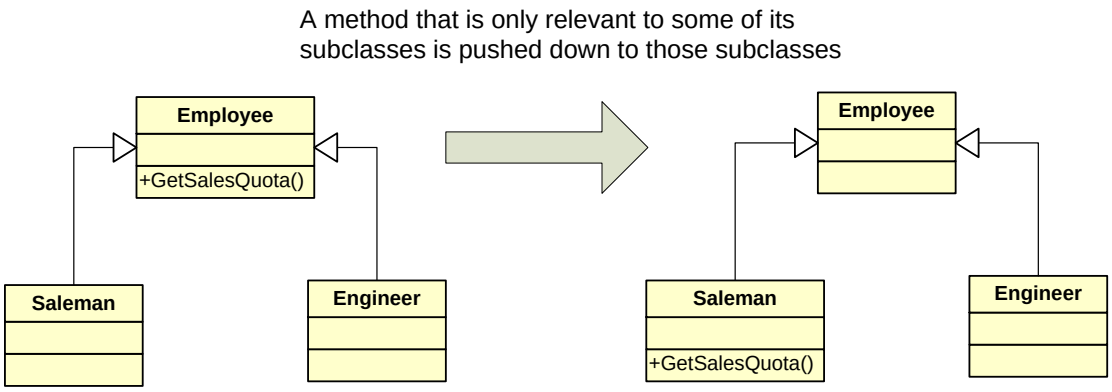

FIGURE 2-1 PUSH DOWN METHOD REFACTORING

The process of refactoring is defined as the following activities (Mens & Tourwe, A survey of software refactoring, 2004):

1. Identify the software to be refactored
2. Select an appropriate refactoring or refactorings
3. Guarantee behaviour preservation
4. Apply the refactoring or refactorings
5. Assess the effect of the refactoring or refactorings
6. Maintain software artifacts to preserve consistency

For the remainder of this section we consider each of the refactoring activities, defined by (Mens & Tourwe, 2004) and repeated above in turn and discuss the related research trends.

The identification of areas of software in need of refactoring has been described as a manual process by Fowler (Fowler M. , 1999) through the detection of *smells*; Fowler describes 22 smells that a developer should look out for and suggests a series of refactorings to eradicate the smells. Mantyla continued Fowler's work on smells by categorising the smells based on their behaviour (Mantyla, 2003). Manual detection of smells, and hence selecting where to refactor software, can become cost prohibitive when dealing with software on a large scale; in this situation it is also much more difficult to identify key areas of software that would benefit from refactoring (Munro, 2005). For example, the eradication of a smell in a particularly fault-prone, heavily reused or complex area of the software may bring a more significant benefit to the

software than the eradication of an identical smell in another area of code. Identifying these problem areas of software remains predominantly a manual process despite acceptance of the need for tool support (Munro, 2005). Currently, tool support is almost exclusively in the area of applying refactorings rather than detecting the need for refactoring (Mens & van Deursen, 2003). Because of this, there has been increasing research in the area of automatically detecting areas of software that would benefit from refactoring and identifying code smells. Balazinska et al., developed a clone analysis tool that detected one well-known code smell: duplicated code (Balazinska, Merlo, Dagenais, Lague, & Kontogiannis), although there have been some negative results with the use of such a tool when compared to a developer identifying where duplicate code should be removed (Zhao & Hayes, 2006). In some cases it was shown that the tool made a decision that a developer would normally not remove duplicate code to the extent that comprehension of the software is detrimentally affected (Mens & van Deursen, 2003). In other cases, it was suggested that the use of a tool for such detection detracts from the experience of the developer and this in itself could lead to a negative impact on the software in the future (Zhao & Hayes, 2006).

Zhao and Hayes use metrics to attempt to predict and prioritise the classes that were most in need of refactoring (Zhao & Hayes, 2006). Their tool applied metrics to locate design problems within the classes and presented them to the developer in priority order. Fenton (Fenton & Pfleeger, 1997) demonstrated how metrics could be used at all stages of the software lifecycle to benefit software and Basili (Basili, Briand, & Melo, 1996) and Olague et al., (Olague, Etzkorn, Gholston, & Quattkebaum, 2007) expanded on this to empirically validate that key design metrics. They used evolution of Chidamber and Kemerer's suite (Chidamber & Kemerer, 1994) to show that those metrics could be used as quality indicators to predict fault-prone classes. Some of these design metrics were used by Zhao and Hayes (Zhao & Hayes, 2006) to identify areas in need of refactoring. They concentrated on size and complexity metrics and validated their results by comparing the tools results with the decisions made by Java programmers on the same code base. Some Java programmers tended to consider size and subjective complexity important while others thought dead code or duplicate code was a greater issue (Zhao & Hayes, 2006). However, there was a degree of unanimity between programmers and the tool (Zhao & Hayes, 2006). Significant time was spent by each

programmer in comprehending the code and then more time in addition in identifying code smells (Zhao & Hayes, 2006). The study also showed that programmers were likely to find the 'easy' problems and overlook the more difficult problems (Zhao & Hayes, 2006) all of which supports the need for tool-based identification.

Zhao and Haye's observation that manual refactoring selection by Java programmers tended to overlook the difficult, more complex and large refactorings and concentrate on the easier, smaller refactorings is supported by research by (Counsell, Hassoun, Johnson, Mannock, & Mendes, Trends in Java code changes: the key to identification of refactorings?) and (Advani, 2006) who create tools that mine for manual refactoring data across multiple versions of the same software to show which refactorings have been applied and when. Their research shows that the majority of refactorings applied are simpler refactorings, i.e., ones that change method signatures and program logic rather than large structural changes such as inheritance hierarchies. This suggests that in practice software is not benefiting from the full range of available refactorings; this is either because the developer is not able to quickly identify the large refactoring opportunities or the developer makes a conscious decision to not make the larger refactoring opportunities due to the complexity, effort and added risk. Both of these supports the view that a tools based approach would benefit the refactoring process and there have been attempts to write such a tool, including the work by Dudziak (Dudziak & Wloka, 2002).

After identifying an area of software to refactor the next step in the refactoring process is to select a specific refactoring or a series of refactorings. Advani et al., showed that often a single refactoring cannot be undertaken by itself (Advani, 2006); Fowler identified that some refactorings explicitly state the need for other refactorings as part of a specific refactoring. However, Advani et al., showed that even when this is not the case it is rare for a refactoring to be applied in isolation. So, in most circumstances the developer must select a number of refactorings to solve the problem or eradicate the smell. This may be linked to the results of Munro's study where it was shown that code smells often occur together; relationships between common code smells were identified (Munro, 2005). The activity of selecting which refactorings, as with the identification of the problem area, is one that currently has little tool support. The developer only receives assistance in the selection process from guidelines, such as those documented

in the description of Fowler's code smells (Fowler M. , 1999) where he informally describes how to eradicate a specific code smell by applying a series of refactorings.

Assessing the effect of refactoring and guaranteeing a refactoring preserves original behaviour is being increasingly achieved by the use of automated unit testing frameworks, most commonly based on the xUnit architecture such as NUnit (NUnit.org) for .NET development and JUnit (JUnit.org) for Java development, and is becoming popularised by the increase in test-driven development and agile methods such as extreme programming (Beck & Andres, Extreme Programming Explained: Embrace Change, 2004). The xUnit framework allows a developer to write repeatable test cases against a unit of code in the same programming language as the implementation code.

Because test cases can test both the normal execution path through the unit of production code and exception paths the code for the test cases can accumulate to a significant amount (van Rompaey, Du Bois, Demeyer, & Rieger, 2007) that needs to be maintained with production code. Some studies (van Rompaey, Du Bois, Demeyer, & Rieger, 2007) (van Deursen, Moonen, van den Bergh, & Kok) have shown the amount of test code in a system to be between 30% and 50% of the total code. When the production code is evolved, either through enhancement or fault fixing, the test code is likely to also to be evolved; this is called co-evolution (Siniaalto & Abrahamsson, 2007). This presents the problem that the test code base - which can be as large as the production code base – can quickly become difficult to maintain.

Van Deursen et al., (van Deursen, Moonen, van den Bergh, & Kok) recongnised that test code has different structures and patterns than production code and published test-specific smells and refactorings to reflect this. Rompaey et al., (van Rompaey, Du Bois, Demeyer, & Rieger, 2007; Tokuda & Batory, 2001; Zeiss, Neukirchen, Grabowski, Evans, & Baker, 2006) have extended this to define metrics to detect specific test smells.

Some of the most common test smells are related to the way a test deals with external resources, such as *Mystery Guest* and *Resource Optimism* and when the test case is inadvertently testing more than the intended unit of code, such as *Indirect Testing*. This is a common problem and is essentially due to the test relying on the result of an additional unit of code executed by the tested unit of code, or the test or tested unit of code relying on an external resource, such as file, database or web service. An

alternative way to deal with dependencies while writing tests is to use *mocks*; mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive. There are several mock testing frameworks that have been developed to allow mocks to be defined and used in testing such as jMock (JMock.org) and EasyMock (EasyMock) for Java and NMock (NMock) for .NET.

Further research into test profiles have concentrated on the classification of tests and how tests are related to the production code. Galli et al., (Galli, Lanza, & Nierstrasz, 2005) noted that there is a varied style and granularity of tests and this can lead to confusion for the developers, in particular: what methods are tested by which tests, to what degree are they tested, what to take into account when refactoring tests and methods and to assess the value of a test. Galli et al., present a taxonomy of tests and provide a tool that attempts to automatically categorise tests and discuss how to link a test with production code methods. The taxonomy includes categories such as *One-method commands, Pessimistic one-method* and *Optimistic one-method*. Their research show that the majority of tests written are *One-method commands*. Galli et al., evaluated IDE support for tests and conclude that while there is support for the execution of tests, normally via the xUnit integration within the IDE, there is little support for further integration. Test integration which provides the following functionality appears to be overlooked in current common IDEs: tighter integration of method and unit test in the IDE; test case selection – automatic selection of relevant tests; concrete typing; test case refactoring – automatic deleting for corresponding tests if the method is deleted, renaming, parameter adding. Further work into this area could include the formalization of test classification, such as including the test type and its relation to the production code in the definition of the test. This would enable the tighter integration between test and production code in the IDE.

Applying refactorings is one area where there is significant tool support (Munro, 2005). Most implementations provide developers with available refactoring options from their current location in the code and present the developer with a preview of the effect of the refactoring before the developer has to commit to making the code change. One significant limitation is that tool support is normally limited to the current code "project"; in large-scale software systems, source code is likely to span multiple code projects which cannot be opened in a single instance of the IDE and this can lead to

refactorings only being applied in localized areas. In the worst case, this can cause failing builds as interfaces referenced from one project to another become unsynchronized.

The final activity in refactoring is to maintain consistency with other software artifacts, such as UML, documentation and schemas. Again, there is little tool support for this area but it does highlight the point that refactoring does not necessarily have to be applied to source code: refactoring can also be applied, for example, to UML at design time (Munro, 2005).

2.3 DESIGN PATTERNS

Design patterns are general reusable solutions to commonly occurring problems in software design. Object-oriented design patterns are a description or template showing relationships and interactions between classes and objects. Gamma et al., (Gamma, 1995) documented 23 design patterns and these have been added to and modified by several others since (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996).

An example of a design pattern is the state design pattern (Gamma, 1995). The state design pattern allows the alteration of an object's behaviour when its state changes.

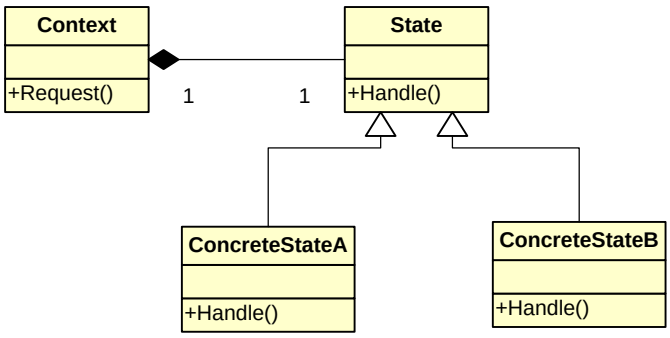

FIGURE 2-2 THE STATE DESIGN PATTERN

Figure 2-2 shows the class structure for the state design pattern. Each state is represented by a different concrete class. Calling *handle* on the state will provide different behaviour depending on which *state* (i.e., which concrete state class) the *context* has.

Many design patterns including those described by Gamma et al., (Gamma, 1995), promote adaptability by supporting modification through specialization. A system built using design patterns can be modified by creating new concrete classes with desired functionality rather than by direct modification of existing classes. We might therefore expect that design pattern participant classes would have a relatively low propensity for change relative to other classes. A previous study by Bieman et al., showed that the opposite was true: design pattern participants were more change-prone than non-pattern classes. Bieman et al., (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003) selected thirteen of the twenty-three design patterns and explored whether their intended benefits and, in particular, whether their relative stability compared to other classes were being empirically realized. In the same study, it was shown the affect this had on change in five commercial, open-source systems along with other contexts of design such as size and inheritance. The study by Bieman et al., in was itself a replication of an earlier study (Bieman, Jain, & Yang, 2001) where 39 versions of a C++ system were used. Results of that earlier study showed that large classes were found to be the most change-prone and that pattern-based classes were more change-prone. Although not specifically looking at change or fault-proneness, there has been some work into automatic extraction of patterns using metrics and other tools (Antoniol, Fiutem, & Cristoforetti, 1998), (Grand, 2002), (Gyimóthy, Ferenc, & Siket, 2005) . Vokac (Vokac, 2004) found significant differences in the fault-proneness of different design patterns in a study of a large C++ industrial system. Aversano et al., (Aversano, Canfora, Cerulo, Del Grosso, & Di Penta) analysed how frequently patterns were modified in three evolutionary open-source systems, the changes they underwent and which classes "co-changed" with the patterns; patterns more suited to support the application purpose tended to be changed more frequently. Work on aspects by (Aversano, Cerulo, & Penta, 2009) also identified 'scattering' and a correlation with design pattern defects. Di Penta et al., (Di Penta, Cerulo, Guéhéneuc, & Antoniol, 2008) explored design pattern 'roles' in three open-source systems; concrete advice on appropriate design was given.

2.4 INHERITANCE CHARACTERISTICS, COUPLING AND COHESION AND SIZE

In the study by Bieman et al., (Bieman & Zhao, 1995), the effect that size and inheritance characteristics had on change in 39 versions of a large C++ system was shown. The

study found that large classes were the most change-prone. Results from a later study by Bieman et al., (Bieman, Jain, & Yang, 2001) using C++ and Java systems were largely inconclusive with respect to class size and change-proneness. Only for two of the systems were large classes more change-prone. The same study also observed counter-intuitive characteristics of the inheritance hierarchy; classes at level zero (the root of a hierarchy) were changed more often than classes at level 1 and 2 of the inheritance hierarchy. In this chapter, we explore the same research question with respect to size and class change addressed in both previous studies.

The role that the 'depth' of inheritance plays in the context of this chapter is highly significant. Many studies have analyzed inheritance in object-oriented systems and most have cast doubt on the use of deep inheritance hierarchies. The Depth of Inheritance Tree (DIT) metric, originally introduced by Chidamber and Kemerer (C&K) (Chidamber & Kemerer, 1994) has been used in many empirical studies investigating inheritance structures. Many studies have reported a lack of use of inheritance to a deep level while others have reported a problem emerging below a certain level. Moreover, only limited numbers of studies have explored the relationship between DIT and faults. Basili et al., (Basili, Briand, & Melo, 1996) was one study that used the C&K metrics as predictors of fault-prone classes. Data from eight medium-sized C++ management systems were collected. Statistically significant results suggested that a class located deep in the inheritance hierarchy (given by its DIT) was more fault-prone than a class higher up in the hierarchy; the study suggested that extensive use of inheritance could have had the opposite effect to that of aiding the maintenance process. Prechelt et al., (Prechelt L. U., 2003) suggested that maintenance effort was positively associated with inheritance depth (i.e., the deeper the inheritance hierarchy, the more maintenance effort required – and this would suggest that this is where the potential for faults to be invested lay). Wood et al., (Wood, 1999) advise that inheritance should be used with care and only when needed. Bieman and Zhao (Bieman & Zhao, 1995) describe a study of nineteen C++ systems, comprising 2,744 classes in total. They found that only 37% of the systems had a median class inheritance depth greater than one. Cartwright and Shepperd (Cartwright & Shepperd, 2000) describe the collection of a subset of the C&K metrics from a large telecommunications subsystem (133,000 lines of C++) and reported relatively little use of inheritance in the system analyzed. However, when it did

occur they found a positive correlation between DIT and number of user reported problems, casting doubt on the use of deep levels of inheritance. The lack of adherence to 'expert' advice on the use of inheritance is further noted in the work of (Gorschek, Tempero, & Angelis, 2010) in a large-scale study of object-oriented practitioners.

In coupling and cohesion studies, Briand et al., (Briand, Daly, & Wust, 1997), (Briand, Daly, & Wust, 1999) presented a unified framework for the measurement of coupling and cohesion in object oriented software systems, comparing a number of measurements of coupling and cohesion and their respective usefulness, integrating existing measurements that examine similar concepts in different ways and proposing a more rigourous decision making process for selecting metrics for a specific goal of measurement. Cain et al., (Cain & McCrindle, 2002) showed that unmanaged coupling within a software project can reduce team productivity and Du Bois et al., (Du Bois, Demeyer, & Verelst, 2004) suggest a set of refactorings to improve the coupling and cohesion of existing code.

2.5 CHANGE AND FAULTS

In studies of fault data in large scale commercial software systems, Hamill et al., (Hamill & Goseva-Popstojanova, 2009) identified common trends in software fault data in two real-world software systems and showed that software faults were often a result of multiple faults spread throughout a complex system; a significant number of faults occurred in late-cycle activities. Ostrand et al., (Ostrand, Weyuker, & Bell, 2005) developed a negative binomial regression model to predict the most fault-prone source files in a future release, based on the fault and change activity of the file in previous releases. However, they did not explore pattern-based classes as part of the study. Arisholm et al., (Arisholm & Briand, 2006) supported Ostrand's earlier findings, showing the relevance of historical change and fault data in predicting future faults in a new release. In further fault studies, Gymothy et al., (Gyimóthy, Ferenc, & Siket, 2005) executed a study to determine how useful the Chidamber and Kemerer object oriented metrics (Chidamber & Kemerer, 1994) were as a mechanism to predict fault-proneness in open source software. They studied several versions of the Mozilla open source project and used fault history stored in the fault tracking software to compare predicted fault-proneness with the actual fault-proneness of the software. Nagappan et al.,

(Nagappan & Ball, 2005) presented a method for early prediction of defect density based on defects found using a static analysis tool. Nagappan showed a strong correlation between the static analysis defect density and the pre-release defect density determined by testing. Sinha et al., (Sinha, Sinha, & Rao, 2010) presented a new approach for determining the origins of a fault, based on the effects a fault fix has on program dependencies and based on those effects attempted to determine the previous fault-introducing version.

3 CHAPTER 3 – REFACTORING

3.1 INTRODUCTION

In the previous chapter, we explored work related to the research areas of our study. We described research in refactoring, testing, design patterns, inheritance, coupling and cohesion. In this chapter, we document an empirical investigation into the first of these aspects of object-oriented software development: refactoring.

Our research objectives for this chapter are:

1. To understand how refactorings are applied in commercial software. Are refactorings regularly applied and, if so, what types of refactorings are most common?
2. To establish whether any refactorings are inter-dependent (that is, if refactoring *x* is applied, is there a likelihood that refactoring *y* will also be applied?)
3. To determine if application of refactoring in commercial software differs from reports from other open source studies?
4. What impact does test code have on the refactoring process?

Refactoring is the process of improving the internal characteristics of existing software while preserving its external behaviour (Fowler M. , 1999). Opdyke first documented the process and potential benefits of refactoring in (Opdyke, 1992), followed later by Fowler's text documenting seventy-two common refactorings (Fowler M. , 1999). Refactoring has been suggested as a positive influence on the long-term quality and maintainability of software and is also said to help prevent (and even reverse) code decay.

In this chapter, we describe the results from applying a bespoke source code analysis tool to 270 incremental versions of WebCSC described in Section 1.4. We examined each version with its predecessor to detect application of 15 types of refactorings in both production and test code classes. The extracted data was analysed based on 1038 separate refactorings extracted by the tool.

The remainder of the chapter is organised as follows. In the next section, we describe preliminaries such as the system studied and the refactorings extracted by the tool. We then analyse the data (Section 3.3), before discussing the threats to the validity of the study (Section 3.4). Finally, we conclude in Section 3.5.

It should be noted that the basis of the research contained in this chapter was published in (Gatrell & Counsell, 2009). The study presented in this chapter therefore represents a replication of an earlier study of open source software by Advani et al. (2006). The express intention of the study presented was to determine if similar refactoring characteristics in proprietary software could be identified.

3.2 METHOD

We studied WebCSC for a two-month period over 270 versions. WebCSC was subject to modifications due to new enhancements as well as fault-fixing. In this chapter, we make no distinction between the two types of maintenance modification (although we do explore these two aspects in later chapters). The WebCSC system was developed in a continuous integration environment where every change committed to source control by a developer was explicitly versioned and built by the continuous integration server; this allowed us to compare code at each version step. On average, a version was submitted to the source control system every hour over the period studied. A significant portion of the source code comprised (unit) test classes and we include these as well as production classes as part of our analysis.

3.2.1 EXTRACTED REFACTORINGS

The tool attempted to detect occurrences of fifteen specific refactorings (in keeping with the earlier study by (Advani, 2006)). The choice of the fifteen refactorings in that earlier study was based on the views of two industrial developers as to the most likely refactorings that developers would undertake as part of their daily maintenance. For convenience, the order of the fifteen refactorings in the following list is *as per the study* of (Advani, 2006), and represents the least frequently applied refactoring (Encapsulate Downcast) to the most frequently applied (Rename Field) identified in that earlier study:

3.2.1.1 ENCAPSULATE DOWNCAST (ED)

According to (Fowler M. , 1999) 'a method returns an object that needs to be 'downcasted' by its callers'. In this case, the downcast is moved to within the method and the methods return type is changed. Figure 3-1 shows the state of a class before and after an encapsulate downcast refactoring has been applied.

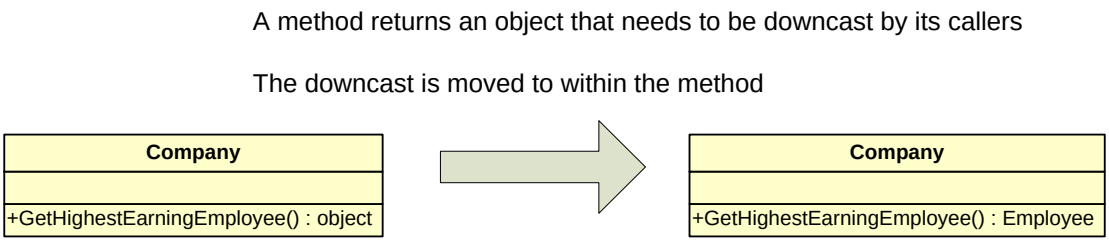

FIGURE 3-1 ENCAPSULATE DOWNCAST REFACTORING

3.2.1.2 PUSH DOWN METHOD (PDM)

'Behaviour on a superclass is relevant only for some of its subclasses' (Fowler M. , 1999). The method is moved to those subclasses. Figure 3-2 shows the state of a class structure before and after a push down method refactoring has been applied.

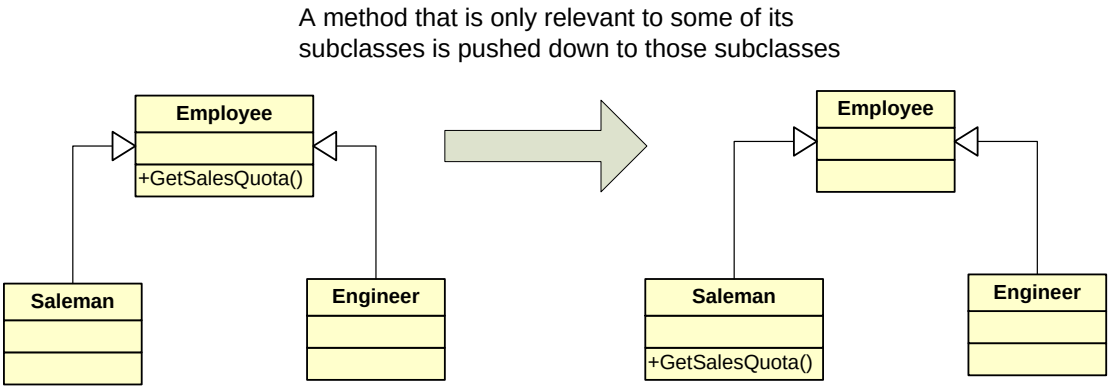

FIGURE 3-2 PUSH DOWN METHOD REFACTORING

3.2.1.3 EXTRACT SUBCLASS (ESUB)

'A class has features that are used only in some instances' (Fowler M. , 1999). In this case, a subclass is created for that subset of features. Figure 3-3 shows the state of a class structure before and after an extract subclass refactoring has been applied.

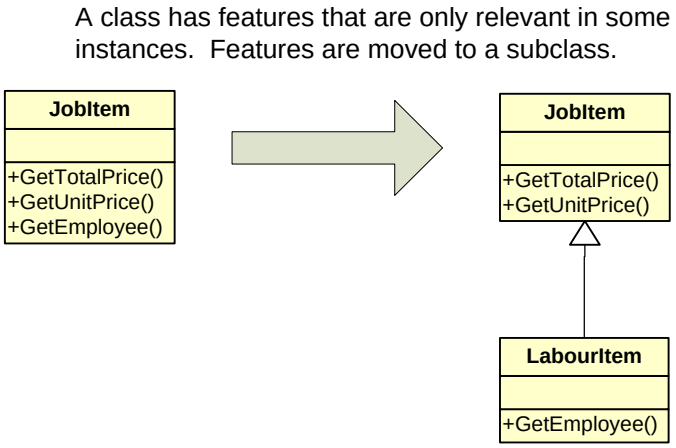

FIGURE 3-3 EXTRACT SUBCLASS REFACTORING

3.2.1.4 ENCAPSULATE FIELD (EF)

The declaration of a field is changed from public to private. Figure 3-4 shows a class before and after an encapsulate field refactoring has been applied.

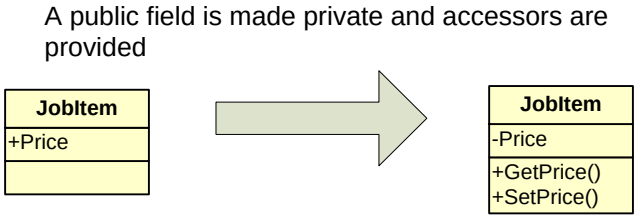

FIGURE 3-4 ENCAPSULATE FIELD REFACTORING

3.2.1.5 HIDE METHOD (HM)

'A method is not used by any other class' (Fowler M. , 1999) (the method should thus be made private). Figure 3-5 shows a class before and after a hide method refactoring has been applied.

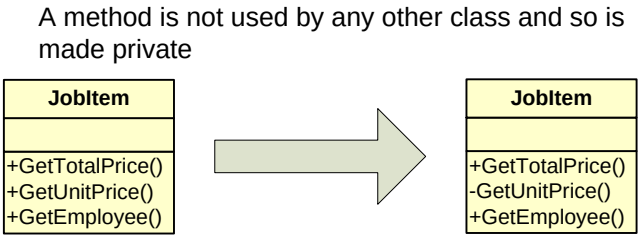

FIGURE 3-5 HIDE METHOD REFACTORING

3.2.1.6 PULL UP FIELD (PUF)

'Two subclasses have the same field'. In this case, the field in question should be moved to the superclass. Figure 3-6 shows the class structure before and after a pull up field has been applied.

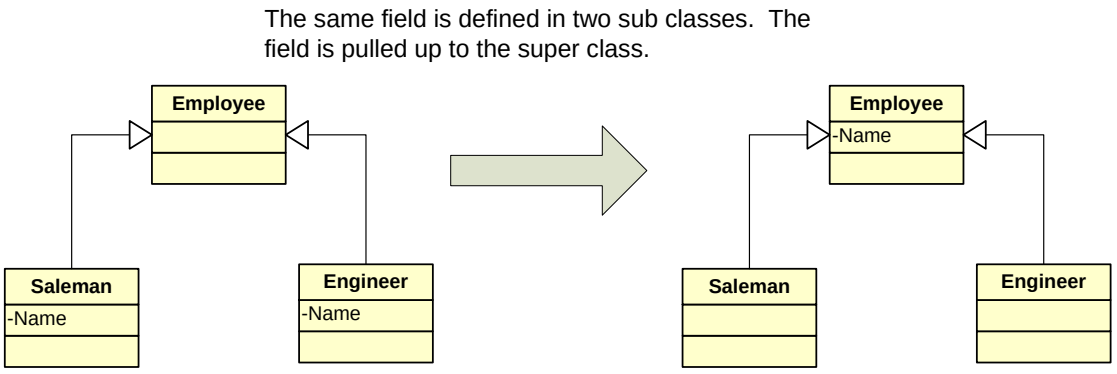

FIGURE 3-6 PULL UP FIELD REFACTORING

3.2.1.7 EXTRACT SUPERCLASS (ESUP)

Two classes have similar features. In this case, a superclass is created and the common features moved to the superclass. Figure 3-7 shows the class structure before and after an extract superclass refactoring has been applied.

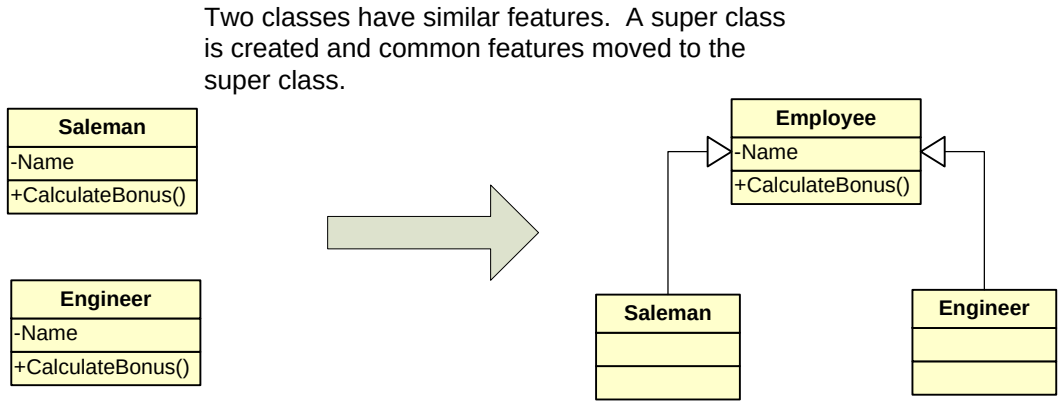

FIGURE 3-7 EXTRACT SUPERCLASS REFACTORING

3.2.1.8 REMOVE PARAMETER (RP)

A parameter is removed from the signature of a method. Figure 3-8 shows a method before and after a remove parameter refactoring has been applied.

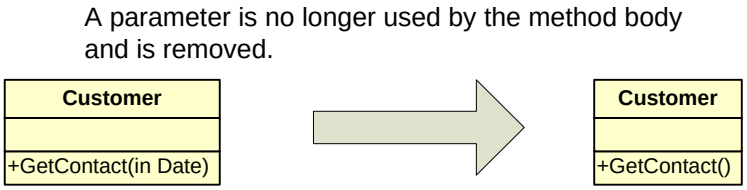

FIGURE 3-8 REMOVE PARAMETER REFACTORING

3.2.1.9 PUSH DOWN FIELD (PDF)

'A field is used only by some subclasses' (Fowler M. , 1999). The field is moved to those subclasses. Figure 3-9 shows the class structure before and after a push down field refactoring has been applied.

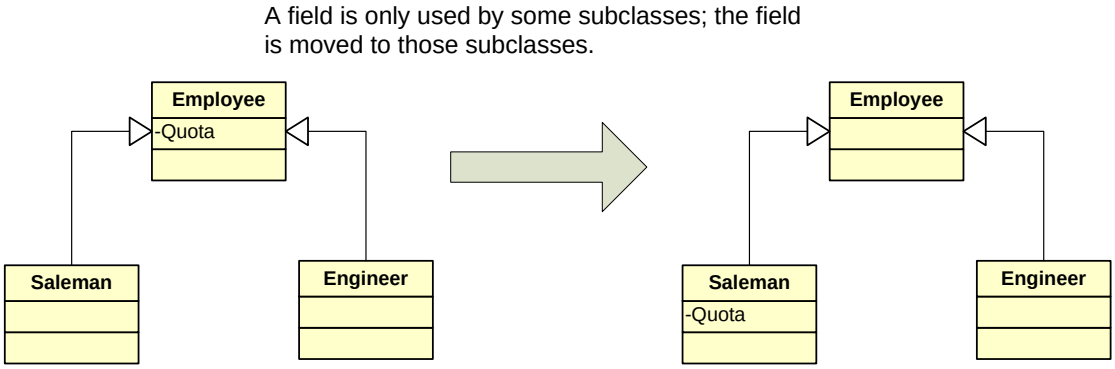

FIGURE 3-9 PUSH DOWN FIELD REFACTORING

3.2.1.10 PULL UP METHOD (PUM)

Methods have identical results on subclasses. In this case, the methods should be moved to the superclass. Figure 3-10 shows the class structure before and after a pull up method refactoring has been applied.

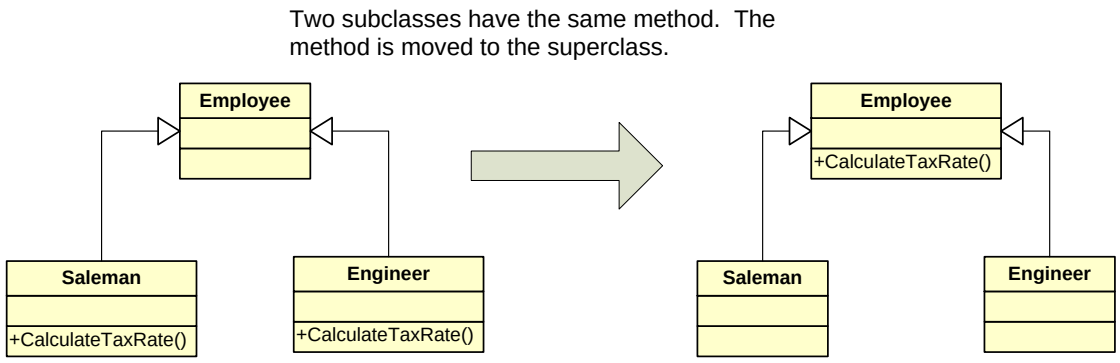

FIGURE 3-10 PULL UP METHOD REFACTORING

3.2.1.11 MOVE METHOD (MM)

'A method is, or will be, using or used by more features of another class than the class on which it is defined' (Fowler M. , 1999). Figure 3-11 shows the class structure before and after a move method refactoring has been applied.

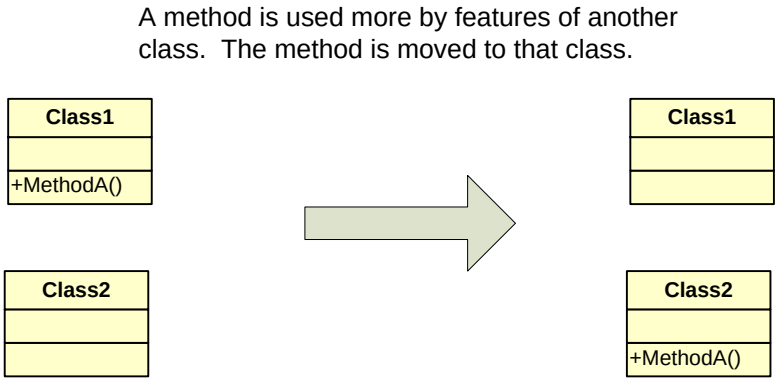

FIGURE 3-11 MOVE METHOD REFACTORING

3.2.1.12 ADD PARAMETER (AP)

A parameter is added to the signature of a method. Figure 3-12 shows a method before and after an add parameter refactoring has been applied.

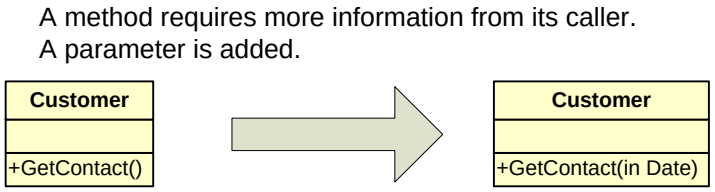

FIGURE 3-12 ADD PARAMETER REFACTORING

3.2.1.13 MOVE FIELD (MF)

'A field is, or will be, used by another class more than the class on which it is defined' (Fowler M. , 1999). Figure 3-13 shows the class structure before and after a move field refactoring has been applied.

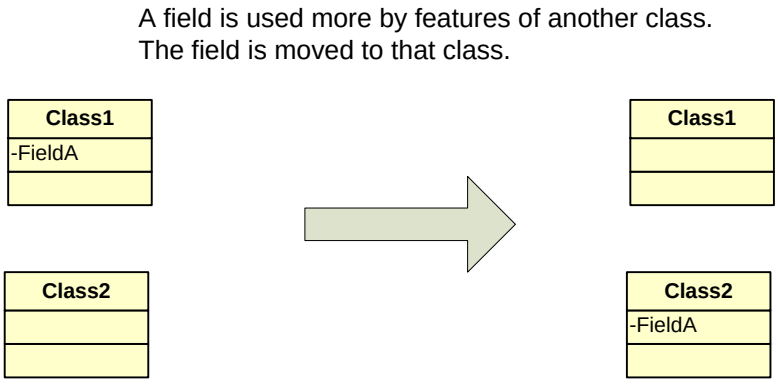

FIGURE 3-13 MOVE FIELD REFACTORING

3.2.1.14 RENAME METHOD (RM)

A method is renamed to make its purpose more obvious. Figure 3-14 shows a method before and after a rename method refactoring has been applied.

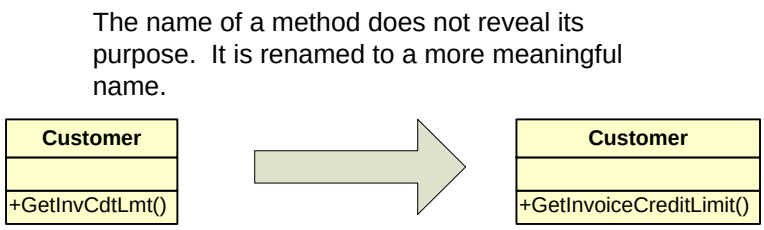

FIGURE 3-14 RENAME METHOD REFACTORING

3.2.1.15 RENAME FIELD (RF)

A field is renamed to make its purpose more obvious. Figure 3-15 shows a class before and after a rename field refactoring has been applied.

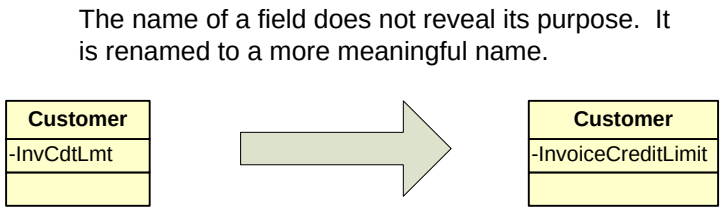

FIGURE 3-15 RENAME FIELD REFACTORING

3.2.2 TOOL MECHANICS AND VALIDATION

To extract each of the fifteen refactoring types, the source data comprising class names, namespaces, method names, method signatures, parameters, fields and return types were stored in a database allowing a) multiple source versions to be loaded and compared at the same time and, b) queries to be run looking for the symptoms of a refactoring. For example, one heuristic was that if a field existed in class A in version *x*, and was removed from class A in version *x* + 1 (and subsequently added to class B in version *x* + 1), an occurrence of a Move Field refactoring was noted. The tool was validated against a set of test classes written specifically to test the output of the tool. Across the test classes, multiple instances of all 15 refactorings were introduced alongside other non-refactoring activity. The test passed if all refactorings were identified by the tool. A sample of the WebCSC classes was then randomly selected to use as input to the tool; this sample was manually inspected to verify that the refactorings identified were correct.

The refactoring tool consists of a user interface (allowing parameters to be set, such as the source location, and the results viewed), a database to store transient data and the results, and a set of detection rules written in SQL as stored procedures.

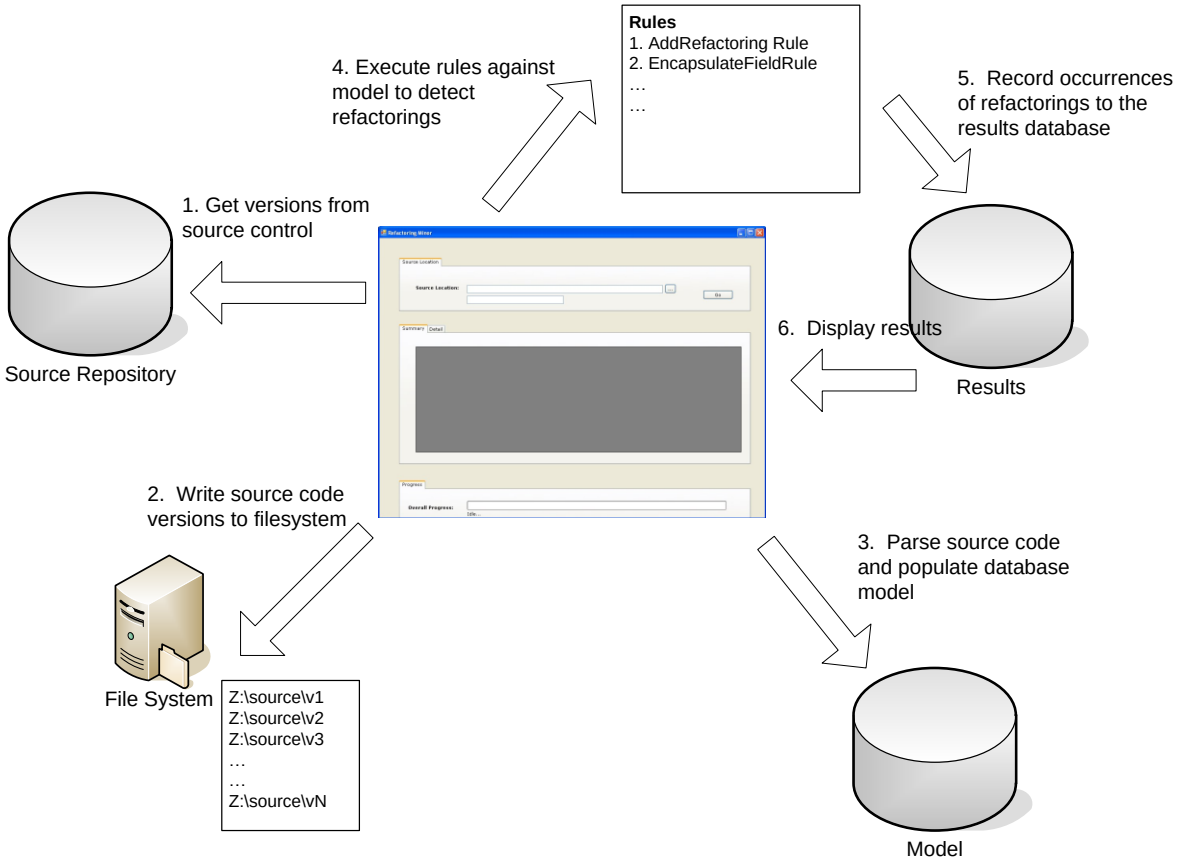

FIGURE 3-16 OVERVIEW OF THE REFACTORING TOOL PROCESS

Figure 3-16 depicts the overall process of the refactoring tool. The first step is to extract the source code of each version to be analysed and place it into a folder on the file system; the folder name indicates the version. The tool is then pointed to the file system location (step 2) and, version-by-version, the source code is parsed and pertinent data populated into the database model (step 3) representing the version. Figure 3-17 shows the database model, storing class, method, method parameter and field data for each version. Once the database model has been populated, a set of rules is executed (step 4) against the model to detect instances of refactorings. There is a separate rule for each refactoring type and each rule is written in SQL as a stored procedure. If one or more refactorings are detected by a rule, the refactoring type, subject and version are returned and these are recorded in the results database (step 5). The results database is also shown in the schema in Figure 3-17. Finally, in step 6, the refactoring tool interrogates the results database and displays all the refactorings found for all versions in the UI.

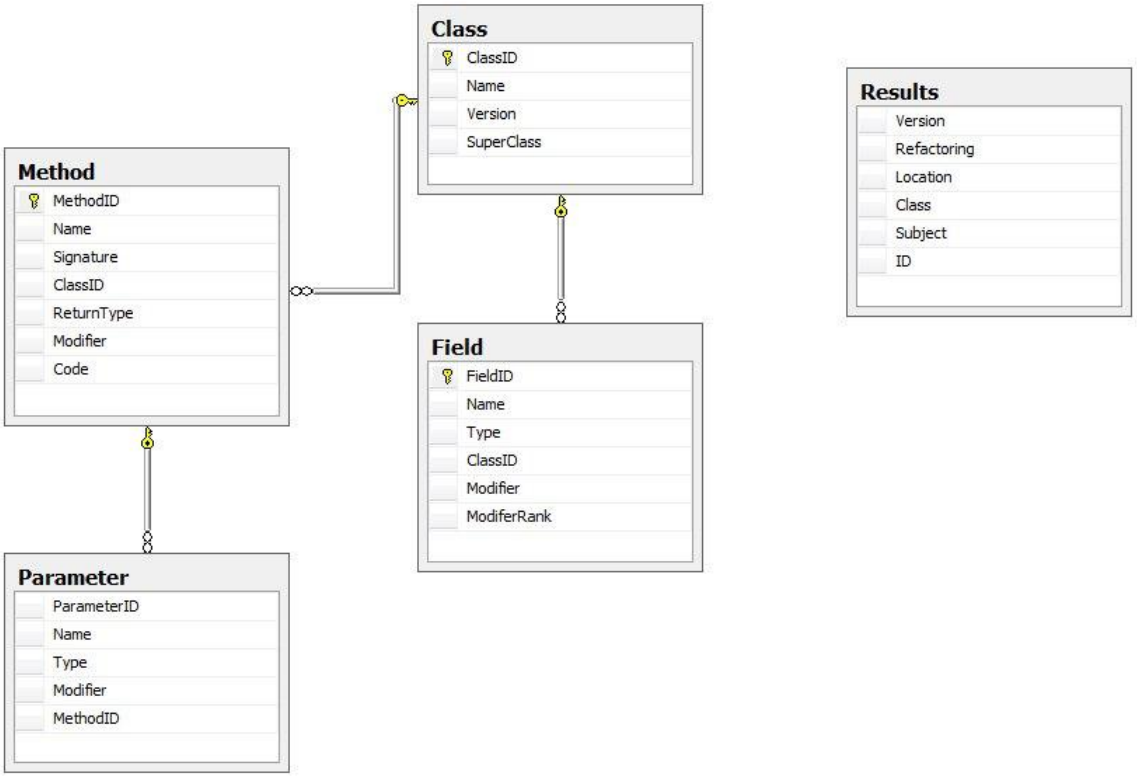

FIGURE 3-17 DATABASE SCHEMA FOR THE REFACTORING TOOL

The refactoring tool is constructed in five main class structures shown in the class diagram in Figure 3-18:

1. The ReportTool class is the main UI and allows certain parameters to be set, controls the execution and displays the results.
2. The CodeDOM component is responsible for reading the source code files from the file system and converting them into an in-memory model.
3. The Model class is responsible for populating the database model representing the parsed source code.
4. The [RefactoringName]Refactoring classes are responsible for executing the rule to detect instances of refactorings from the database model representing the parsed source code, by calling the corresponding stored procedure.
5. The Results class is responsible for recording any detected refactorings into the results database and also reporting back the full list of results.

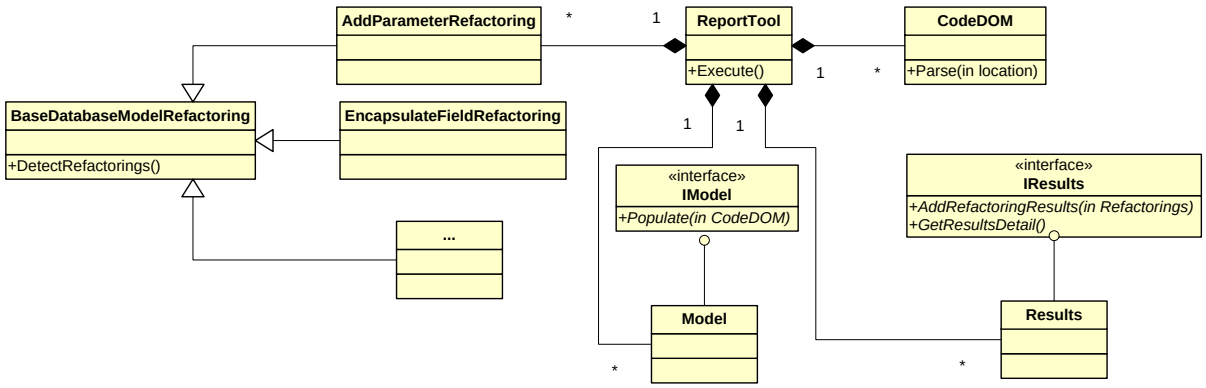

FIGURE 3-18 CLASS DIAGRAM OF THE REFACTORING TOOL

Figure 3-19 is a sequence diagram showing the interaction between the core classes of the refactoring tool.

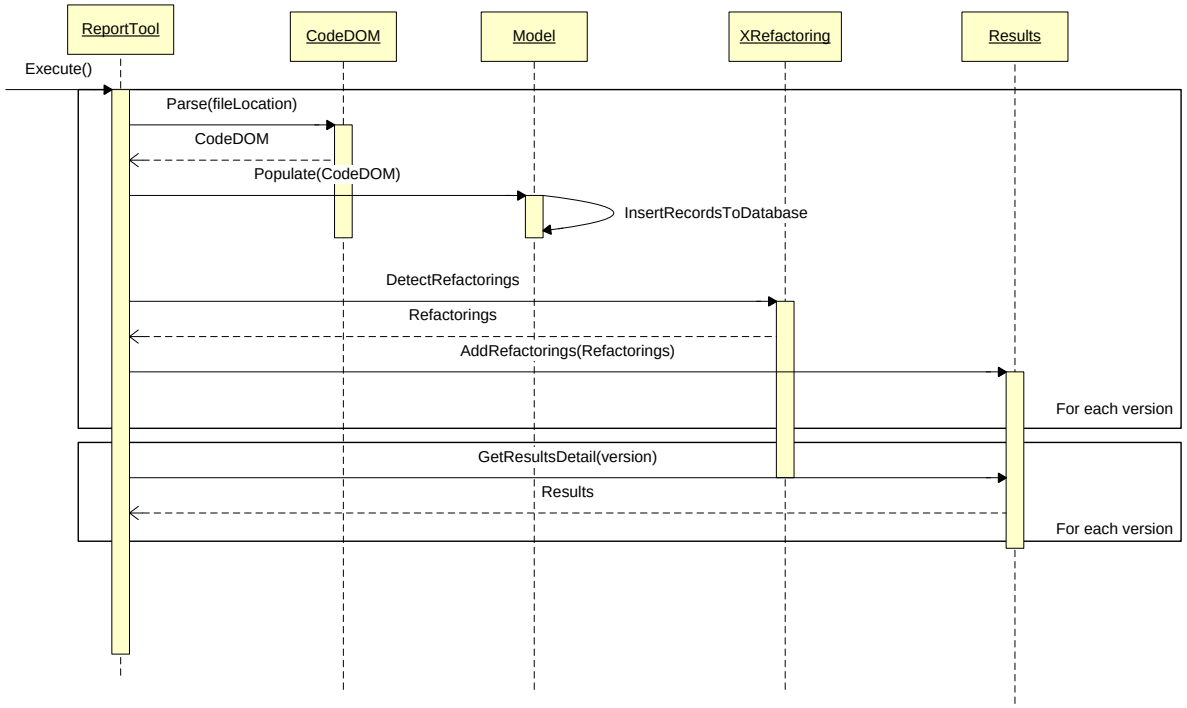

FIGURE 3-19 SEQUENCE DIAGRAM OF THE REFACTORING TOOL

Table 3-1 describes the rules that were defined to detect the refactorings when comparing one version of the source code to the next. These rules were converted to

SQL statements and executed against the database model (Figure 3-17) representing the source code versions as part of the tools detection process.

Refactoring	**Rule for Detection**
Encapsulate Downcast	An encapsulate downcast refactoring exists if a method exists in version *n* + 1 also exists in version *n* but with a different return type (in the parent hierarchy).
Push Down Method	A push down method refactoring exists if a method exists in class a in version *n* + 1, the same method did not exist in class *A* in version *n*, the same method did exist in the parent class of class *A* in version *n* and the same method did not exist in the parent class of class *a* in version *n* + 1.
Extract Subclass	An extract subclass refactoring exists if class *A* has a child class (say, class *B*) in version *n* + 1, class *B* does not exist in version *n*, at least one field or method from class *A* in version n exists in class *B* in version *n* + 1 (and is removed from class *A* in version *n* + 1).
Encapsulate Field	An encapsulate field refactoring exists if a field that exists in version *n* + 1 also exists in version *n*, but is defined with a higher modifier rank in version *n*. *Public* is the highest modifier rank, followed by *protected;* finally *private* is the lowest rank.
Hide Method	A hide method refactoring exists if a method that exists in version *n* + 1 with a *private* modifier exists in version *n* with a *public* modifier.
Pull Up field	A pull up field refactoring exists if a field exists in version *n* + 1 in class *A*, does not exist in class *A* in version *n*, exists in at least one sub class of class *A* in version *n*, and does not exist in the same class in version *n* + 1.
Extract Superclass	An extract superclass refactoring exists if two classes exist in version *n* + 1 with the same superclass, the same two classes in version *n* do not have the same superclass, at least one feature of the two classes in version *n* are moved to the new superclass in

	version *n* + 1.
Remove Parameter	A remove parameter refactoring exists if a method exists in version *n* + 1 with at least one less parameter than in version *n*.
Push Down Field	A push down refactoring exists if a field exists in class *A* in version *n*, does not exist in class *A* in version *n* + 1, does not exist in a subclass of class *A* in version *n*, and does exist in at least one subclass of class *A* in version *n* + 1.
Pull Up Method	A pull up method refactoring exists if a method exists in two subclasses sharing the same parent class in version *n*, and the method does not exist in those subclasses in version *n* + 1, and the method does exist in the parent class in version *n* + 1.
Move Method	A move method refactoring exists if a method exists in class *A* in version *n*, does not exist in class *B* in version *n*, no longer exists in class *A* in version *n* + 1, and does exist in class *B* in version *n* + 1.
Add Parameter	An add parameter refactoring exists if a parameter exists for a method in version *n* + 1, but does not exist for the same method in version *n*.
Move Field	A move field refactoring exists if a field exists in class *A* in version *n*, does not exist in class *B* in version *n*, no longer exists in class *A* in version *n* + 1, and does exist in class *B* in version *n* + 1.
Rename Method	A rename method refactoring exists if a method exists in version *n*, and a method (that did not exist in version *n*) with the same code, parameters, and return type exists in version *n* + 1, but with a different name.
Rename Field	A rename field refactoring exists if a field exists in version *n*, and a field (that did not exist in version *n*) with the same modifier and type exists in version *n* + 1, but with a different name.

TABLE 3-1 RULES TO DETECT REFACTORINGS

3.3 DATA ANALYSIS

In this section we explore the results of empirical investigation. First, we look at the number and types of refactorings that were detected. Second, we compare the refactoring trends to those seen in previous open-source refactoring studies. Third, we look for any inter-dependencies between common refactorings. Finally, we explore the impact that test code has on the refactoring process.

3.3.1 REFACTORING FREQUENCY

During the two-month period incorporating the 270 code versions, a total of 1038 refactorings drawn from the set of 15 refactoring types were detected by the tool from 196 classes. Of the 270 versions, 94 incorporated at least one refactoring; no refactorings were therefore applied in 176 versions. The highest ranked 30 classes from the 196, in *descending* number of applied refactorings, accounted for 51.15% (531) of the total number of refactorings. A high proportion of classes had had only a single or two refactorings applied to them over the course of the 370 versions.

Figure 1 illustrates the distribution of this data. The class with the most number of applied refactorings (52) was a test class; in fact, only 13 of the highest ranked 30 classes were production classes - test classes underwent relatively larger amounts of refactoring than the corresponding production classes. The explanation for such a high level of refactoring in these classes, verified by the WebCSC project manager was that for every production class refactoring, up to five changes have to be made to the test class responsible for testing that production class. This was an interesting result and demonstrates the synergy and co-evolution of test and production classes.

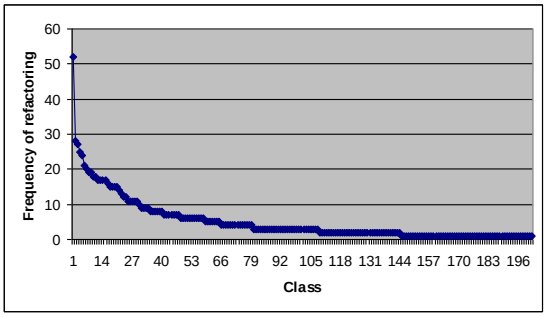

FIGURE 3-20 FREQUENCY OF REFACTORING IN CLASSES

Table 3-2 shows the frequency breakdown of refactorings. Six classes had had between 21 and 30 refactorings applied and 23 classes had had between 10 and 19 applied refactorings. The high bias towards classes having 1 or 2 refactorings can be seen from the table where 87 classes fell into this category.

>= 30	>= 20	>=10	> 2 < 10	= 1 <= 2
1	6	23	79	87

TABLE 3-2 FREQUENCY OF APPLIED REFACTORINGS

3.3.2 ANALYSIS THEMES

The data analysis in this chapter is explored through three themes. First, we explore any potential similarities between the number and type of refactorings found in our study and that found in the previous study by (Advani, 2006) where open-source software was used as a basis. Second, we explore the emphasis of, and comparison between, refactorings in test code and those in production code to establish whether any significant differences or patterns could be found. Finally, since the tool identified both the class *and* method where each of the 1038 refactorings were undertaken, we investigate whether specific 'related' refactorings were applied in unison to the same class or method. An open research question at present is whether application of one refactoring implies that at least one other related refactoring also has to be applied at the same time so our study seeks to inform that open research question.

3.3.3 STUDY COMPARISON.

Figure 3-21 shows the data for the fifteen refactorings extracted by (Advani, 2006) from multiple versions of seven OSS in ascending (and same) order on the *x*-axis as *per* the list given in Section 3.1. For example, the least frequently applied was refactoring 1 (Encapsulate Downcast) and the most frequently applied refactoring 15 (Rename Field). Figure 3-21 shows the values extracted for WebCSC in the same order as in Figure 3-22.

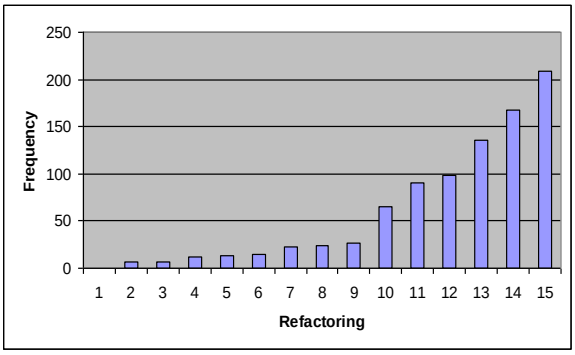

FIGURE 3-21 REFACTORINGS TAKEN FROM (ADVANI, 2006)

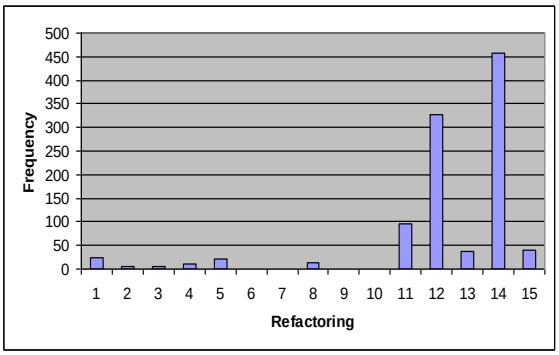

FIGURE 3-22 REFACTORINGS TAKEN FROM WEBCSC

For the WebCSC system, we see that over the 270 versions studied there was a relatively high occurrence of five individual refactorings (i.e., refactorings 11-15 from Figure 3-22). When compared with the study by (Advani, 2006), we observe a strong correspondence. In that earlier study, a 'Gang of Six' refactorings were identified (i.e., refactorings 10-15 from Figure 3-21: Pull Up Method, Move Method, Add Parameter, Move Field, Rename Method and Rename Field). Concurring with that earlier study, five of those refactorings in the WebCSC system *also* seem to have been applied relatively frequently. The more 'complex' structural refactorings, on the other hand, such as Extract Subclass (refactoring 3) and Extract Superclass (refactoring 7) both of which require structural changes to the inheritance hierarchy were rarely applied to either the earlier study by (Advani, 2006), *or* to the WebCSC system. In fact, no occurrences of the Extract Superclass refactoring were found in the WebCSC system and only 4 Extract Subclass refactorings were found in total across all 270 versions. For the WebCSC system, 25 Encapsulate Downcast (ED) refactorings were identified by the tool, but in the earlier study by Advani et al., not one of these refactorings was identified in any

version of the seven systems. Inspection of the raw data revealed the majority of the ED refactorings to occur exclusively in production classes (rather than test classes).

The most common refactoring from Figure 3-22 is refactoring 14, Rename Method (RM). Surprisingly, the whole set of RM refactorings occurred in only 50 of the 270 versions (i.e., a large number of renaming of methods occurred in a comparatively small number of classes). In version 154, 201 of the total of 458 RM refactorings were found to have been applied. Figure 3-23 shows the profile of the RM refactoring across the versions of the WebCSC system; the flattening trend at version 154 is clearly visible and striking.

Consultation with the WebCSC project manager revealed why version 154 comprised so many of the RM refactorings. The work being undertaken on the set of classes in which those RM were applied had been 'checked out' and had been worked on for a relatively longer period of time than the set of classes changed in other versions. In other words, the high number of RM refactorings reflected the amount of time that the relevant classes had been 'out of the system' and given special attention. It is also interesting that nine of the 23 Encapsulate Downcast refactorings for the WebCSC system shown in Figure 3-22 were applied in version 154. This was due to the large number of renaming methods in the same version, the signatures of which were affected when an ED refactoring was undertaken. This provides some evidence of the relationship between these two refactorings.

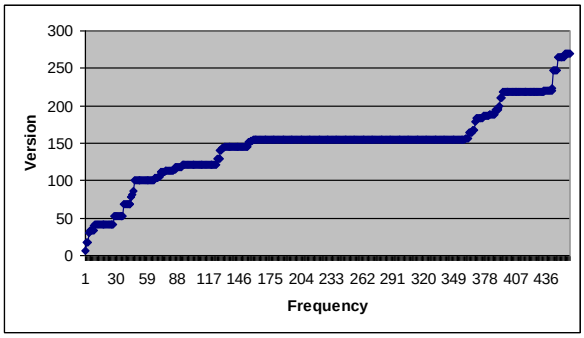

FIGURE 3-23 PROFILE FOR RENAME METHOD

Figure 3-22 also shows that 328 Add Parameter (AP) refactorings were undertaken over the course of the 270 versions. Figure 3-24 shows the version profile for this

refactoring and indicates a relatively consistent application of AP over the course of the 270 versions (in contrast to the graph in Figure 3-23 where version 154 experienced a relatively large number of RM refactorings). Addition of parameters is a standard change made frequently to test classes in the face of regular changes and addition of functionality to the set of production classes. Consequently, the AP refactoring was frequently and regularly applied to test classes.

Typically, a WebCSC test class includes test methods for testing a) every parameter in the production class b) parameter exception conditions (for example, null checks) c) outcomes of the method d) every exception condition and e) every execution path that has not already been tested. Multiple changes to a production class usually induce multiple changes to the corresponding test class.

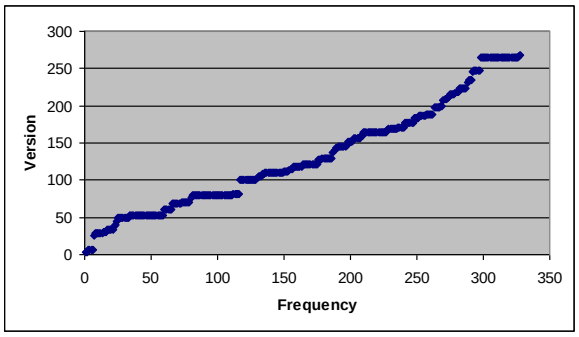

FIGURE 3-24 PROFILE FOR ADD PARAMETER

3.3.4 TREND ANALYSIS.

A feature of the earlier analysis by (Advani, 2006) was the tendency for significant refactoring activity to occur, followed by relatively quiet periods of inactivity.

Henceforward for clarity of analysis, we decompose the versions of WebCSC into separate time periods.

Figure 3-25 shows the analysis for WebCSC for the first 50 versions and Figure 3-26 the number of refactorings for versions 51-200. There is a clear trend for refactorings to increase steadily between versions 1 and 40, and moreover, in an erratic fashion. The peak of 16 refactorings at version 42 comprised entirely of Move Method (MM) and RM refactorings, unique to a set of test classes. In the case of the same 16 refactorings,

visual inspection of the data revealed that many methods *moved* were those that were then subsequently *renamed*. In the WebCSC system, if a production class method is moved and renamed, then this requires the corresponding test class method also be moved and renamed to retain the 'semantic' mapping.

One question that arises from the preceding analysis is whether, during the three stated periods (versions 1-50, 51-200 and 201-270), and particularly the middle (versions 51-200), the system saw a rapid growth in either number of classes or methods. We raise this question because we would usually expect a relatively high rise in number of refactorings to accompany a corresponding addition of functionality (classes and methods) to a system. Between versions 1 and 50, only 21 classes were added to the system; between versions 51 and 200, 167 classes were added and between versions 201 and 270, only 35 classes were added. This suggests that addition of classes may have been the impetus for extra required refactoring effort noted in the middle interval in this case. We also note that the corresponding three periods saw a rise of 215, 1177 and 225 methods, respectively - further supporting the view that a relatively large increase in classes and associated methods induced correspondingly large numbers of refactorings. In terms of the first goal of the study, we therefore found strong parallels between the study of (Advani, 2006) and that of the study presented.

Figure 3-26 shows the refactoring data for versions 51 to 200 and shows a similar effect to that in Figure 3-25. Remarkably, from Figure 3-25, the average number of refactorings *per* version (sample 1-50) was 1.92, compared with 5.34 *per* version from Table 3-3. The average then fell to 2.1 between versions 201 and 269 (Figure 3-27). In other words, the earlier versions of the system saw relatively small, yet fluctuating amounts of refactoring and so too did the later versions; the interval in between, however, saw a large rise in refactoring activity. In terms of numbers, between versions 1-50 there were only 92 refactorings; that rose to 801 between versions 51-200 and 145 between versions 201 to 270

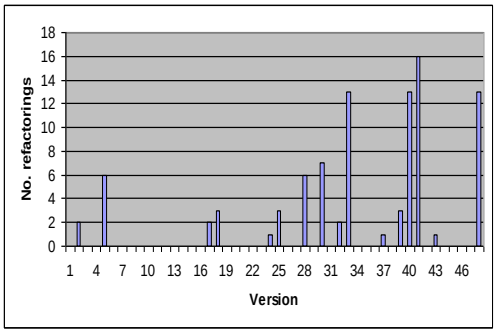

FIGURE 3-25 REFACTORING ACTIVITY FOR WEBCSC (FIRST 50 VERSIONS)

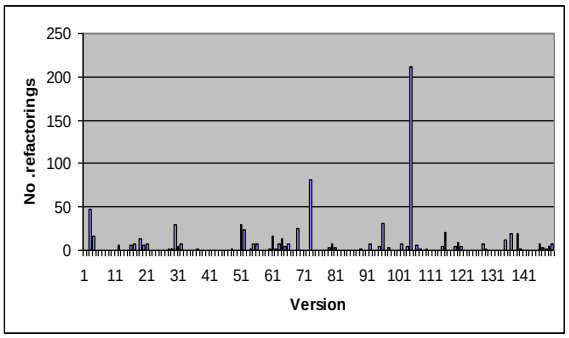

FIGURE 3-26 REFACTORINGS (VERSIONS 51-200)

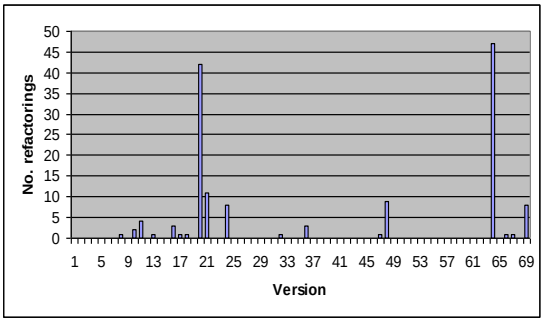

FIGURE 3-27 REFACTORINGS (VERSIONS 201-270)

	Refactoring	**Test**	**Prod.**
1.	Encapsulate Downcast	6	19
2.	*Push Down Method*	1	3
3.	*Extract Subclass*	1	3
4.	Encapsulate Field	3	7
5.	Hide Method	17	4
6.	*Pull Up Field*	0	1
7.	*Extract Superclass*	0	0
8.	Remove Parameter	2	10
9	*Push Down Field*	0	1
10.	*Pull Up Method*	0	0
11.	Move Method	75	22
12.	Add Parameter	107	221
13.	Move Field	3	35

14	Rename Method	255	203
15.	Rename Field	4	35

TABLE 3-3 REFACTORINGS (TEST AND PRODUCTION CLASSES)

The general lack of inheritance-based refactorings (all of which are italicised in Table 3-3) is a feature of both test and production classes. The absence of inheritance-based refactorings in test classes is due to the fact that test classes are usually independent, self-contained units and have a one-one mapping between with the production classes they test. As such and unlike production classes, there is no requirement or motivation to either inherit behaviour or take advantage of inheritance features from other test classes.

The relatively large number of Hide Method refactorings in the test code is an exceptional situation and is the result of a single developer using temporary methods in a test class which were then changed from public to private once those methods had been used (the project manager confirmed this to be the case in follow-up discussions). One final feature of the set of test class refactorings is that there are relatively few RF refactorings (refactoring 15), in complete contrast to production classes. A simple explanation accounts for this feature again supported by the follow up discussions with the project manager; in test classes, very few fields (i.e., attributes) are generally declared, since the class has no obvious need to store or manipulate data. The importance of refactoring test classes is signified by the fact that overall, 474 of the 1038 refactorings were found in the test code (approx. 46% of the total).

3.3.4.1 VERSION ANALYSIS.

A further aspect of any difference between test code and production code is to what extent the application of refactorings were applied in test code and production code *in the same versions*. In other words, when a refactoring was applied to production code, was at least one refactoring applied to test code? The preceding discussion implies that the two may go hand-in-hand. To investigate this feature of the data, we computed the number of versions where a) there was at least one test code refactoring *and* at least one production code refactoring, b) where there were *only* production code refactorings and finally, c) where there were *only* test code refactorings.

The data revealed that in 34 versions there was: 'at least one test refactoring *and* one production class refactoring'. In 46 versions, only production code refactorings were undertaken and finally, in 14 versions, only test class refactorings were undertaken. The majority of the versions where there were only production class refactorings tended to be single, isolated refactorings. This contrasts with occasions when only test class refactorings were applied, when there was usually a cluster of the same type of refactoring.

While there is some evidence to suggest that test and production refactorings are combined in certain versions (and these have been highlighted in the preceding discussion), there is also evidence of an independence of each type of refactoring (given by the fact that they were applied unilaterally in different versions). One interesting observation from the data that might explain this feature was that a large number of test classes were refactored in the version *immediately following* a large group of production class refactorings. In other words, developers may, in some cases, apply refactorings to production classes and then, in the next version, make the necessary test class refactorings. This evidence of co-evolution between production and test code supports the results of (Zaidman, 2008) and this tendency of developers in the WebCSC system was supported by the project manager in the follow-up discussions. Knowledge that a group of test class refactorings usually follow production class refactorings would be useful knowledge for a project manager in allocating tasks and resources.

3.3.5 RELATED REFACTORINGS.

One aspect of refactoring that we also wanted to explore was the tendency for a combination of different types of refactorings to be applied at each version. We might expect some refactorings operate in pairs, for example, if we move a method, then we might reasonably expect to have to move a field at the same time. Equally, we might also expect a method to be renamed after having to move it. The mechanics specified by Fowler identify the relationships between refactorings and these relationships have been previously explored in (Advani, 2006). In this section we determine if the same related refactorings discovered in Advani's study of open-source software occur in WebCSC.

Figure 3-28 shows the graph of number of Move Method refactorings on a version-by-version basis and the number of Move Field refactorings when at least one Move Method was identified (taken from the earlier study of Advani et al., (Advani, 2006)). Figure 3.29 shows the corresponding data for refactorings from the WebCSC system where a strong correspondence between the two graphs can be seen; the relationship is thus preserved in the WebCSC system.

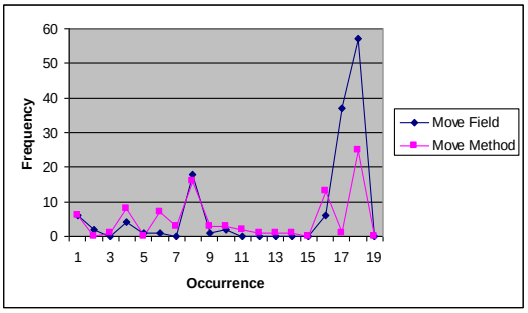

FIGURE 3-28 RELATIONSHIP - MF AND MM (ADVANI, 2006)

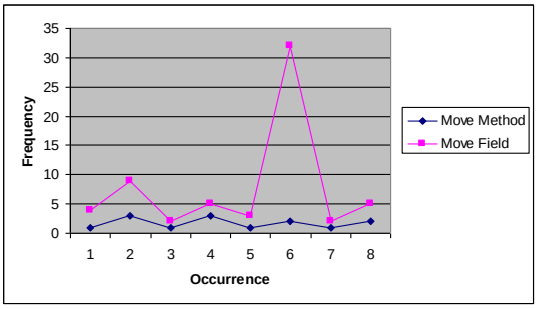

FIGURE 3-29 RELATIONSHIP - MF AND MM (WEBCSC)

A further relationship we might expect is that between RF and RM (for example, if a field is renamed, it is not beyond the bounds of possibility that an associated method performing an operation on the renamed field may also be renamed to retain consistency).

Figure 3-30 shows the relationship extracted from the data for every occasion where 'at least one RF and at least one RM were identified' in the data extracted by the tool. Only on 12 occasions were both RM and RF applied in the same version. Although a strong parallel between the two refactorings exists, the sum of RM refactorings from Figure 3-30 is 78 from a total of 458; the sum of RF on the other hand was 24 from a total of 39.

In other words, RF is more closely tied to RM than *vice versa*. One reason for this may be the existence of *get* and *set* methods in the respective classes and where we might expect the relationship to be unidirectional only; that is, if a field is renamed, the two methods that provide read and write access to that field (the *get* and *set* methods) are likely to change also to retain consistency.

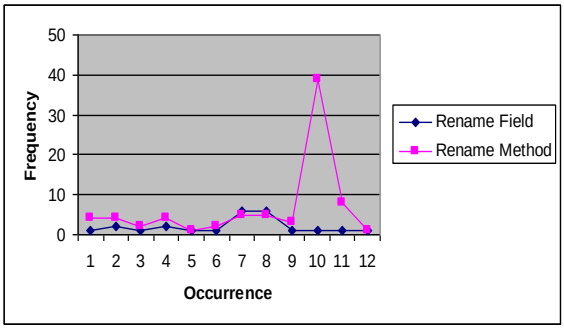

FIGURE 3-30 RENAME FIELD AND RENAME METHOD

3.3.6 TEST VERSUS PRODUCTION CODE

The tool identified refactorings in both test code and production code. One question that we explored was the extent to which refactorings (and composition of refactorings) in each of these two types of code differed (or concurred) in keeping with the study of (Zaidman, 2008)? Of the 3174 classes in the system at version 270, 1735 (54.6% of the total) were production classes and 1441 (45.4%) test classes. Based on discussions with the WebCSC project manager, corresponding and concurrent fluctuations in refactorings could be expected to have been applied to both test classes and production classes, since changes in one are usually reflected in required changes to the former. We would also expect the type of refactoring in each case to be similar. In the case of both production and test code, we would expect a high proportion of movement and renaming (MM and Rename Field (RF) and RM), as well as many AP refactorings to reflect the need to test features of modified production classes.

3.3.6.1 NUMBER OF REFACTORINGS

Figure 3-31 shows the number of refactorings of the fifteen types in the test classes compared with the same fifteen in the production classes. (The order of the refactorings is the same as that in Figure 3-21 and Figure 3-22.) Table 3-3 shows the actual values taken from this figure.

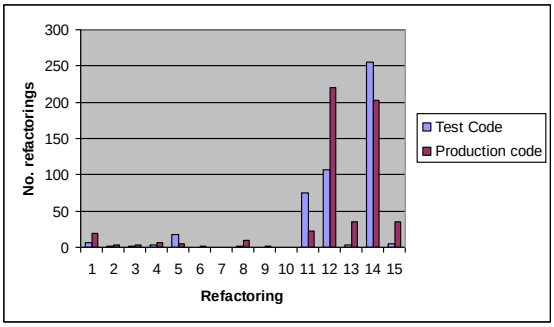

FIGURE 3-31 REFACTORINGS APPLIED TO TEST CODE AND PRODUCTION CODE

3.4 DISCUSSION

A number of threats to the validity of the study need to be considered. First, we have only considered fifteen of the seventy-two refactorings listed by Fowler in his seminal text (Fowler M. , 1999). It is possible that many more refactorings could have been applied over the versions of WebCSC studied. However, the fifteen refactorings chosen represent a broad cross-section of those seventy-two covering object-oriented themes such as inheritance, encapsulation as well as standard tasks that we would expect a developer to undertake as part of their programming role (i.e., moving and renaming class features). To implement all seventy-two refactorings is a task that would require development by a competent programmer over many years. We see our study as the 'first-step' towards understanding the intricacies of the refactoring process across environments.

Second, the WebCSC language was written in C# and the study upon which we based our work was written in Java, comprising multiple systems. Further, the environment of WebCSC was proprietary and not OSS, the development process was continuous integration and the time frame for C# system was only one year. In defence of these potential criticisms however, only minor differences exist between Java and C#. A major motivation for the work was to determine if trends in OSS refactoring occurred in proprietary software *and* OSS. It is exceptionally difficult to obtain free-to-use, evolution-based, commercial software and while we do not claim the work to be a replication in the truest sense of the word, we feel that it is important that results from previous studies are tested, especially for an emerging discipline like refactoring. We also feel that it is important to evaluate different environments (proprietary versus OSS) even if the development processes are different (i.e., these comparison problems

will *always* exist). Moreover, we could claim that while OSS development at a generic level underpins an ethos of shared and freely-available software, the processes and dynamics of development and people within each OSS system are likely to be very different (itself a threat).

Third, we have reported a similar result to that of (Advani, 2006) and to a lesser extent that of (Zaidman, 2008), and we have provided an explanation of why that phenomenon might occur due to the relationships between refactorings (Section 4.2.6). A wider issue is whether developers 'consciously' undertake refactorings as part of a concerted refactoring effort or they undertake those refactorings as part of more regular maintenance activity (i.e., it happens automatically when fault-fixing). We see this as a topic of future research.

Fourth, we could be criticised for writing a bespoke tool instead of reusing the original one of (Advani, 2006). However, we *did* reuse the algorithms for detecting each of the fifteen refactoring. The original tool mined OSS resources and was geared towards extraction of code and subsequent XML development from sourceforge.net. On the other hand, the data used by WebCSC was already available in an easily converted format. On balance, it would have taken just as long to adapt the original tool than to have re-written it.

Fifth, one plausible reason why so few 'complex' refactorings were unused is simply that the majority of time in any system is spent fixing faults or adding new functionality and not on refactoring; equally, that developers avoid complexity. . Finally, we accept that *regular* changes to the code in WebCSC (aside from the identified refactorings) are just as interesting and worthy of study and comparison with the refactorings identified. However, we leave that as a significant topic for future work.

3.5 CONCLUSIONS

In this chapter we have described an empirical study of refactorings extracted from versions of a large commercial C# system.

Our research aims for this chapter were:

1. To understand how refactorings were applied in commercial software. Are refactorings regularly applied, and if so what types of refactorings are most common?
2. To establish whether any refactorings are inter-dependent (that is, if refactoring *x* is applied is there a likelihood that refactoring *y* will also be applied?).
3. To find out if the application of refactorings in commercial software differ from what we have previously seen in open source studies?
4. What is the impact of test code on the refactoring process?

Fifteen specific types of refactoring were extracted using a bespoke tool. A number of results emerged from the analysis. First, while simpler refactorings were common, more complex structural refactorings were rare in common with a previous study by (Advani, 2006). Second, refactorings were applied sporadically, rather than consistently over the period of time studied again supporting the earlier study. We also found some interesting similarities between test classes and production classes in support of recent work by (Zaidman, 2008). Finally, we suggested a reason why certain refactorings might be applied in unison.

The question that is often asked is whether developers *do* refactoring and if so what are the common types of refactoring? The study presented shows that significant amounts of refactorings had been undertaken. Second, an emerging research area is the refactoring requirements for test suites and the set of refactorings that lend themselves to the way test classes are constructed; we feel we have provided some insights into that area, although we accept there are many more outstanding challenges. The study is thus a contribution to the body of refactoring knowledge of systems on an evolving basis.

4 CHAPTER 4 - THE RELATIONSHIP BETWEEN DESIGN CONTEXT AND FAULT AND CHANGE-PRONENESS

4.1 INTRODUCTION

In the previous chapter we documented an empirical study into refactoring. This highlighted the amount of change that the system under study had undergone during the period under study. It also highlighted the type of changes that the same system had undergone and led to a need to explore more deeply the object-oriented features of that system that might contribute to change (and fault-proneness). In this chapter, we therefore continue with an empirical study into the design context of a class (specifically: inheritance, coupling, cohesion and size) in relation to fault and change-proneness in commercial software.

Our research questions for this chapter are:

1. Are the key aspects of the object-oriented design context: inheritance, coupling and cohesion, and size, contributory factors to fault and change propensity?
2. Are there properties or characteristics of these aspects of the design context that cause fault or change-proneness more than others?

Identifying changes made to a system over time can help identify problem areas and also inform remedial action by the software practitioner. Such data can also be instrumental in helping to predict future change, the prioritization of work and the allocation of limited resources. The same potential benefits are true of fault data in its role of highlighting problematic areas of code and possible directed re-engineering or refactoring effort (Fowler M. , 1999) (Arisholm & Briand, 2006). The extent to which these benefits can be realized is an issue of significant interest for exploratory, empirical software engineering studies. A number of studies have also cast doubt on the extent to which deep levels of inheritance assist with the maintainability of a system (Prechelt, Unger, Philippsen, & Tichy, 2003), (Cartwright & Shepperd, 2000), (Basili, Briand, & Melo, 1996), (Harrison, Counsell, & Nithi, 2000).

In this chapter, we analyze change through the design context of a class. More specifically, we explore change through the size of a class, its inheritance characteristics, and its coupling and cohesion properties. We also explore fault data for the same system and the same design perspectives (size, inheritance, coupling and cohesion).

WebCSC had been subject to 19,054 changes over a two year period of the study - these changes were due to both enhancements and fault fixing. Inheritance properties of each class were identified based on their inheritance depth and the number of subclasses (i.e., children) belonging to each. Coupling was measured using afferent coupling and efferent coupling metrics (described in section 1.3.2, and expanded upon in section 4.2.3) and cohesion was measured using a variant of the Lack of Cohesion in Methods (LCOM) metric (described in section 4.2.4). The size, inheritance coupling and cohesion characteristics are compared to the change history of the classes to determine any relationships. Fault data over a later period is also analyzed to support this analysis related to change-proneness.

The remainder of the chapter is organised as follows. In the next section, we describe preliminaries such as the system studied and the metrics recorded. We then analyse the data (Section 4.2), before discussing the threats to the validity of the study (Section 4.4). Finally, we conclude in Section 4.5.

It should be noted that the basis of the research in this chapter was first published in (Gatrell & Counsell, 2010) and (Gatrell & Counsell, 2011).

4.2 METHOD

The period over which our change analysis is based represents two years of system development. Each modification in the version control system, whether for a fault fix, enhancement (or otherwise) constituted a new version for the class and each version was counted as a single change. For the purpose of this study and to align the study with that of previous studies (Bieman, Jain, & Yang, 2001), (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003), we assume that each change made to code by a developer is equivalent, i.e., the relative size of the change in terms of LOC required by the change is not considered.

4.2.1 SIZE MEASURES

For the purpose of this study, size was measured by Lines Of Code (LOC), Number of instance methods in a class, Number of static methods in a class and Number of fields in a class. We note that LOC only considers executable code. Declarations were not counted, nor were interfaces, abstract methods or enumerations. Comments were also ignored and where a single logical LOC was spread over multiple lines for coding style, e.g., there were a large number of arguments to a method call, only a single LOC was counted. We also collected Total number of methods, Number of properties (defined as 'getters' or 'setters' of a field in C#) and Number of operations (defined as the sum of the number of class fields and methods). These metrics are in keeping with the earlier studies of Bieman et al., (Bieman, Jain, & Yang, 2001), (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003).

A bespoke tool written by the authors was used to extract this information from the WebCSC system - the version control system contained all changes made to every class since its inception.

4.2.2 INHERITANCE PROPERTIES

The inheritance properties of classes were measured through the Depth of Inheritance Tree (DIT) and the Number of Children (NOC) belonging to class metrics. Both of these metrics, originally defined by C&K (Chidamber & Kemerer, 1994), have been used extensively in empirical studies since (Basili, Briand, & Melo, 1996) (Daly, 1996) (Harrison, Counsell, & Nithi, 2000). DIT was collected by considering each class in the WebCSC system and determining the maximum length of the path from the class to its root class. The NOC was collected for each class by determining the number of immediate subclasses (note, we use the term 'subclass' and 'child' inter-changeably in this Thesis). The DIT and NOC metrics were extracted using a plug-in to the build server for the WebCSC system.

4.2.3 COUPLING

We measured both afferent coupling and efferent coupling for each class in the studied system. Afferent coupling measures the number of coupling types that directly depend

on a class. To calculate the afferent coupling for a class in the system: *class X*, the sum of the classes in the system that use *class X* is measured. Efferent coupling measures the number of coupling types that a class directly depends on. To calculate the efferent coupling for a class in the system: *class X*, the sum of the classes in the system that are used by *class X* is measured. They therefore represent opposite views of coupling.

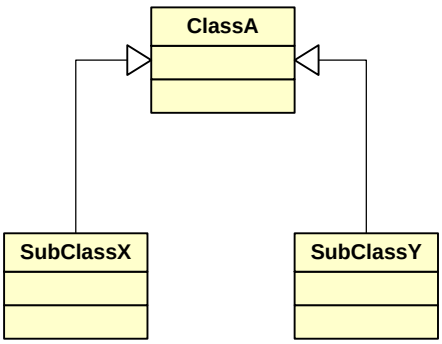

FIGURE 4-1 A CLASS STRUCTURE SHOWING EFFERENT AND AFFERENT COUPLING

To illustrate the difference between afferent and efferent coupling, Figure 4-1 shows a simple class structure: Sub Class X and Sub Class Y both inherit Class A; in this example Class A has an afferent coupling of 2, as Sub Class X and Sub Class Y directly depend on it; Sub Class X and Sub Class Y have an efferent coupling of 1, as they both directly depend on Class A.

The more common Coupling Between Object (CBO) metric of (Chidamber & Kemerer, 1994) measures the number of classes coupled to a class (both efferent and afferent couplings) but does not make any distinction between incoming and outgoing coupling. We wanted to see if there was a distinction between coupling at a low level of granularity when related to fault-proneness and so for this reason we chose afferent and efferent coupling metrics.

4.2.4 COHESION

The cohesion of each class in the system studied was calculated using the Lack of Cohesion in Methods (Henderson-Sellers) metric. The Henderson-Sellers version of the LCOM metric was used as it attempts to overcome the problems identified with Chidamber and Kemerer's (Chidamber & Kemerer, 1994) original LCOM metric (one

such problem results in very different classes having an equal LCOM value). This metric is defined in Figure 4-2 and results in a value between zero and one inclusively:

```
                LCOM(HS) = (M - sum(MF)/F)(M-1)


Where:

  M = Number of methods

  F = Number of fields

  MF = Number of methods of the class accessing a
particular field
```

FIGURE 4-2 LCOM(HS) METRIC

To illustrate we present two concrete examples. The first, of a cohesive class. If a class has five methods and two fields, and every method accesses both fields, the class would have an LCOM(HS) value of: (5 – (10/2))/4 = 0, i.e., it is a highly cohesive class. Our second concrete example shows a uncohesive class: if a class has 5 methods and two fields, but only one of those classes accesses both fields, and the other methods do not access the fields at all, the class has an LCOM(HS) value of: (5 – (2/2))/4 = 1, i.e., an uncohesive class.

4.3 DATA ANALYSIS

During the two year period, a total of 19,054 changes were made to the system. 4,434 of the 7,439 classes had no changes made to them at all over the same period. A large number of classes had between 2 and 30 changes made, and only 29 classes had had 30 or more changes. The most frequently changed class had had 145 changes applied to it, nearly double the number of changes of the second most change-prone class, with 75 changes.

Table 4-1 shows the frequency of changes per class and shows that 29 classes had over 30 changes, 56 had between 20 and 29 changes, and 280 had had between 10 and 19 changes applied to them.

0 changes	1-9	10-19	0-29	>= 30
4,434	2,687	280	56	29

TABLE 4-1 NUMBER OF CHANGES/CLASS

The high bias towards classes having less than ten changes can be seen from Table 4-1; in total, 4,434 classes had had no changes at all applied to them and 2,687 had had between 1 and 9 changes. The mean number of changes per class in the system was 2.55.

4.3.1 HYPOTHESES H1-H3

Three hypotheses were explored as part of our study of changes made to the WebCSC system. We note that the first hypothesis is identical in composition and wording to that originally posed by Bieman et al., in (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003); the second has been changed marginally from that also proposed by the same study to read more succinctly and the third hypotheses is one that we investigate independently. The study presented therefore represents a replication of an earlier study.

Hypothesis H1: Are larger classes more change-prone? A larger class has more functionality and there is therefore a greater likelihood that some functionality in the class will need to be corrected or enhanced.

Hypothesis H2: Classes located high up in an inheritance hierarchy will be more change-prone. Such a class has more dependents and there is therefore a greater likelihood that some functionality in the class will need to be enhanced because of changing requirements in those lower-down dependent classes.

In other words, the use of specialization in an inheritance hierarchy places a responsibility on classes high up in the hierarchy to provide appropriate functionality to subclasses as a part of changing requirements also.

Hypothesis H3: Classes with a large number of children (subclasses) are more change-prone than other classes. This hypothesis is based on the belief that a class with many

children will be the subject of greater maintenance activity, since there are added dependencies on the parent class because of the changing requirements of that large number of children.

4.3.2 CLASS SIZE AND CHANGE (H1)

In this chapter, each of a set of class size measures was correlated against number of changes. Figure 4-3 to Figure 4-9 show the relationship between number of changes and each of those size measures. From inspection of these figures we see that all of: LOC, Number of instance methods, Number of static methods, Number of fields and Total number of operations there is a trend towards greater change-proneness.

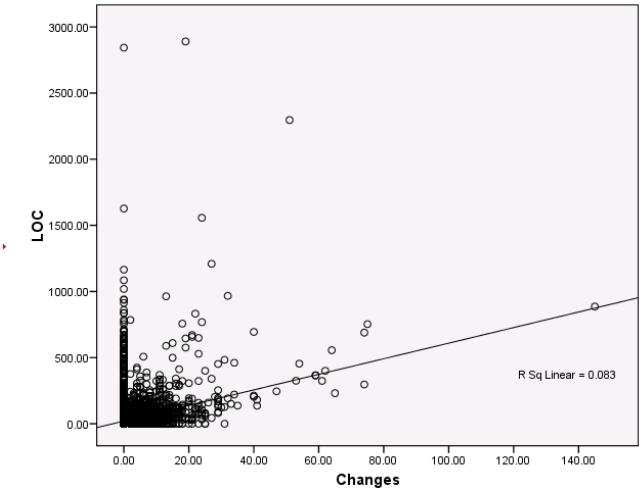

FIGURE 4-3 LOC VS. NUMBER OF CHANGES

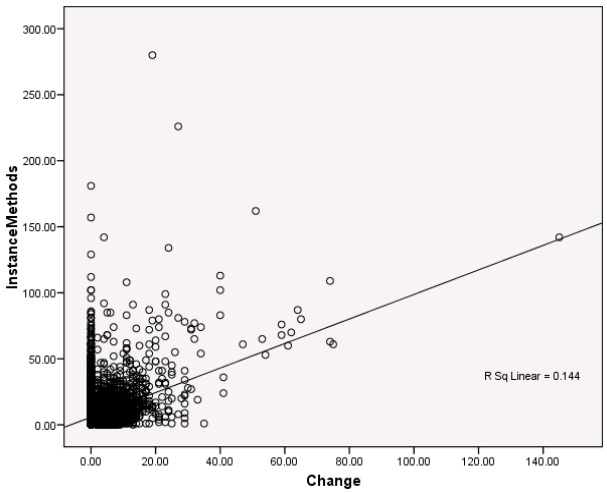

FIGURE 4-4 INSTANCE METHODS VS. CHANGES

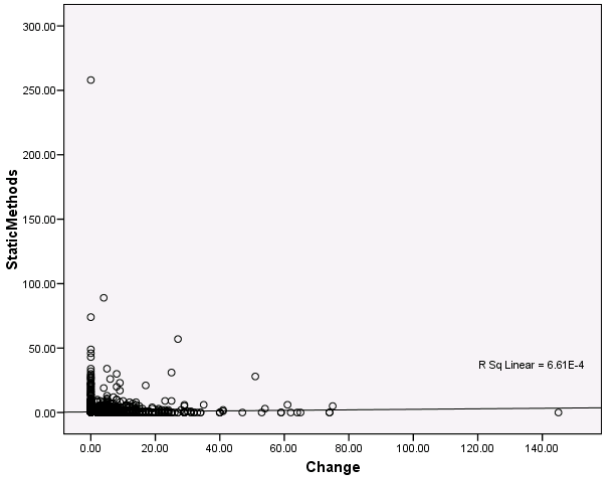

FIGURE 4-5 STATIC METHODS VS CHANGES

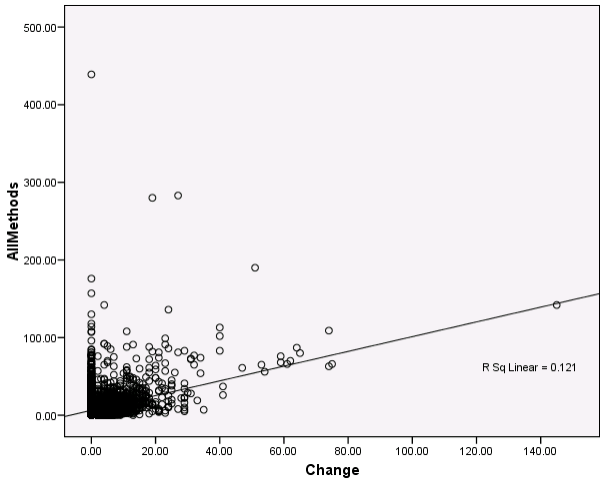

FIGURE 4-6 METHODS VS CHANGES

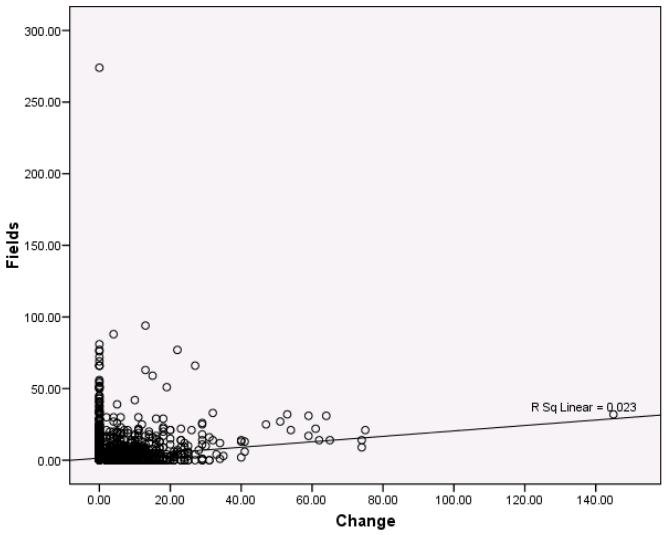

FIGURE 4-7 FIELDS VS CHANGES

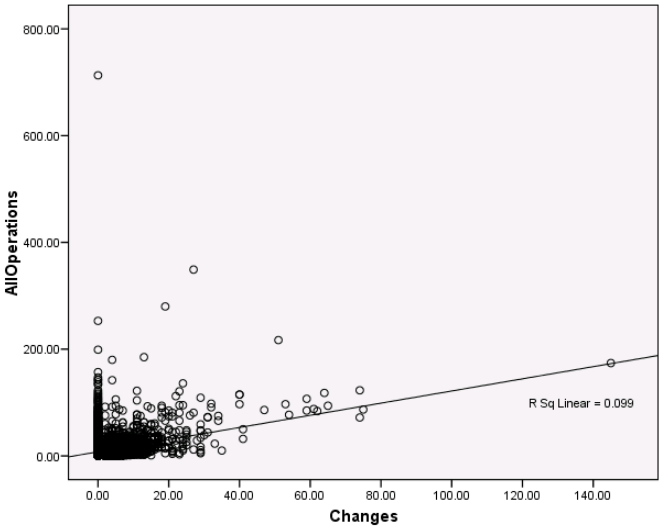

FIGURE 4-8 OPERATIONS VS CHANGES

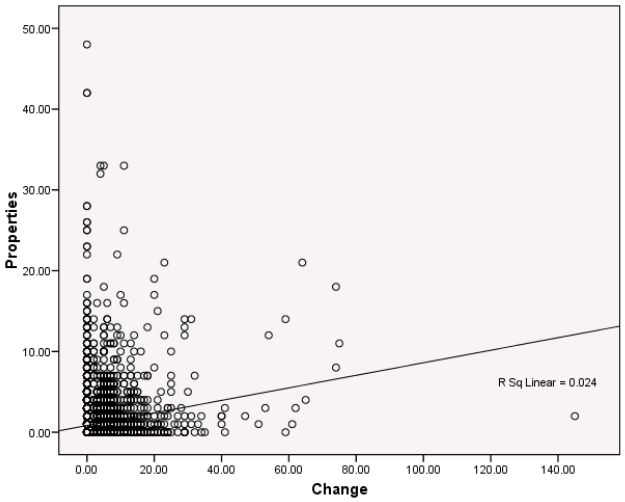

FIGURE 4-9 PROPERTIES VS CHANGES

Table 4-2 shows the statistical correlation values for each class size metric versus change. A single '*' represents statistical significance at the 1% level and a '**' asterisk, significance at the 5% level.

We computed both Pearson coefficients (a parametric test) and Spearman rank (a non-parametric test) correlation coefficients (Field, 2005). The values for Pearson's coefficients show that all size measurements had a strong influence on the propensity for change. Spearman's rank correlation values show that all size measurements (except number of static methods and number of fields) also had a strong relationship with change propensity. The number of static methods has the weakest relationship (Pearson's value of 0.03), while the number of instance methods has the strongest relationship change (Pearson's value of 0.38).

One explanation for the lack of correlation found between static methods and change is that the latter do not actually use instance variables – so, in theory, they may be less likely to be modified as regularly as methods containing instance variables. In other words, we would expect classes where no instance variables are being manipulated to be modified less than classes that do (other things remaining equal). One suggestion for the relative lack of correlation for fields is that it is not often that fields (in either their name or declaration) are changed – it is the functionality that manipulates those fields which we would expect to comprise the bulk of changes made to a class. In other words, what the methods do with those instance variables is (on average) likely to be more subject to change than modification of the declared instance variables themselves.

The strongest correlation (in terms of Spearman's coefficient) was for LOC. Since the more methods, the greater the number of LOC, we would expect an explicit size measure to correlate with any feature of a class. In Bieman et al's study (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003), the relationship between operations and changes was found to be stronger than that between fields and changes and from Table 4-2 the same relationship appears to be the case. In contrast to the earlier study however, where operations were found to have the strongest relationship, our study indicates that a number of features (LOC, instance methods, total methods and properties) all correlate more strongly than that for operations.

Metric	Pearson's	Spearman's
LOC	0.29*	0.18*
Operations	0.32*	0.14*
Instance methods	0.38*	0.16*
Static methods	0.03**	0.00
Methods	0.35*	0.17*
Properties	0.16*	0.17*
Fields	0.15*	-0.02

TABLE 4-2 CHANGE CORRELATION COEFFICIENTS

In terms of the original hypothesis, we can clearly find strong support for H1 in light of the evidence presented. Large classes are more change-prone; however, in keeping with the earlier result reported in (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003) and in contrast with (Bieman, Jain, & Yang, Design patterns, design structure, and program changes: an industrial case study., 2001), we do not find overwhelming support for the hypothesis.

We need to be mindful of the fact that there are size features of a class in the case of the system studied that might reduce the chances of those feature needing to be changed – the role of static methods and fields are cases in point here. We also posit that each system is likely to have its own idiosyncrasies that might cause it to exhibit slightly different correlation features than other systems based purely on the different nature of its application domain. The same is true whether using open-source or proprietary software.

4.3.3 DIT ANALYSIS (H2)

The second hypothesis (H2) we explore is whether 'classes located high up in an inheritance hierarchy are more change-prone than those lower down?' To determine the extent to which inheritance characteristics influenced class change, we again used the class-based Depth of Inheritance (DIT) and Number of Children (NOC) metrics of C&K as a vehicle (Chidamber & Kemerer, 1994).

Figure 4-10 shows the change profile for classes at different DIT levels. The system mean change value appears as a base line value of 2.55. The majority of classes had a DIT range of between 0 and 3. Classes in this category all had a similar rate of change to the mean of change for the entire system. However, classes with a DIT in excess of 3 had a higher rate of change than the system average. Many studies of inheritance have shown that systems typically have a very low median DIT value (Bieman & Zhao, Reuse through inheritance: A quantitative study of C++ software, 1995) (Cartwright & Shepperd, 2000) (Nasseri, Counsell, & Shepperd, 2008). Several studies have also suggested that DIT 3 is the threshold level before inheritance becomes unwieldy and impractical to use (Daly, 1996) (Harrison, Counsell, & Nithi, 2000) and the evidence here seems to support that trend. Figure 4-10 shows that there is a clear peak in the propensity for class change at DIT 5 and 6. In fact, the number of changes rises rapidly after DIT 3, and then peaks at DIT 6, before decreasing again.

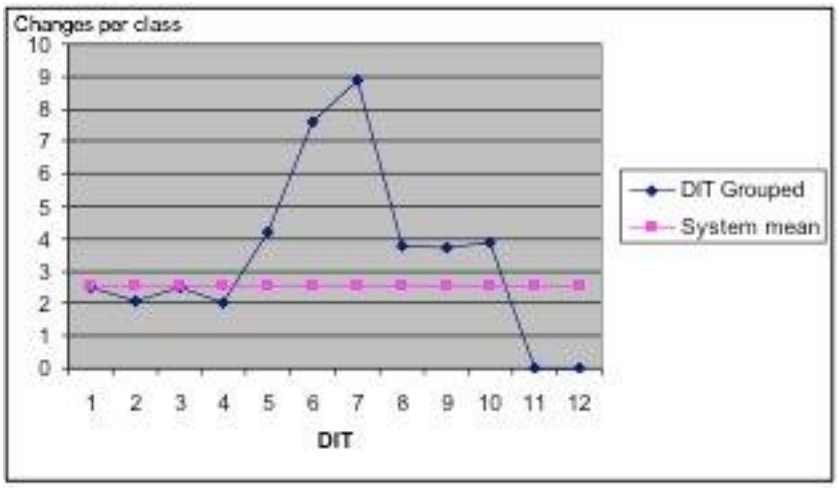

FIGURE 4-10 MEAN CHANGES PER DIT

Figure 4-10 shows graphically the mean changes per class (Mean Change) grouped according to DIT level. The peak of 8.88 can be seen at DIT 6 after which the mean change falls significantly. The result for DIT indicates that there is an interval (the shaded region in Table 4-3) where relatively fewer changes are made to classes both below it and above it.

One criticism of this analysis might be that since the majority of classes are likely to be at DIT 0-3, then that is inevitably where the changes will take place. However, since we are observing mean values, this does not explain the high values at DIT 4-6, or indeed the sudden fall at DIT 7. From the preceding analysis, we find little support for H2 from the data. It is not necessarily true that classes located high up in an inheritance hierarchy are more change-prone. It is actually in the middle tiers of the hierarchy where most changes on average are made. This result would seem to go against current and past thinking in terms of inheritance (embodied in the original hypothesis).

DIT	Classes	Mean change
0	1116	2.65
1	2445	2.14
2	1836	2.64
3	1362	1.98
4	296	4.39
5	92	7.38
6	41	8.88
7	214	3.29
8	52	3.19
9	31	3.52
10	1	0
11	1	0

TABLE 4-3 CHANGES/CLASS GROUPED BY DIT

4.3.4 NOC ANALYSIS (H3)

While the DIT metric provides a useful profile of one aspect of the inheritance hierarchy, it does not provide a view in any sense of the width of the hierarchy. The NOC provides this feature and allows us to establish whether a relationship exists between change and inheritance width.

Table 4-4 shows the data extracted from the system to support this hypothesis. The NOC values collected from the WebCSC system showed that the majority of classes had zero immediate subclasses. Those classes had a similar rate of change to the mean of

the entire system (i.e., 2.55 changes). Classes with an NOC between 1 and 20 however, had a higher rate of change than the mean of the system. Classes with an NOC in excess of 20 were found to have a proneness to change similar to the mean of the system. The largest mean change was for classes with an NOC value of between 11 and 20.In keeping with the findings for DIT therefore, a clear pattern emerges of classes below and above a certain interval of NOC being more change-prone than those either side of that interval. Table 4-4 shows this effect clearly. Statistical analysis showed a Pearson correlation coefficient of 0.03 (significant at the 5% level) and a Spearman rank correlation coefficient of 0.16, significant at the 1% level (for number of changes vs. NOC in each case).

NOC	Classes	Max. Ch.	Mean Ch.
0	5951	145	2.30
1-10	1412	75	3.36
11-20	76	25	4.14
>20	238	74	2.79

TABLE 4-4 CHANGES/CLASS GROUPED BY NOC

In terms of the original hypotheses H3, there is an interval phenomenon being exhibited, outside of which change is fairly consistent. Within that interval however, change is significantly higher. We cannot therefore find support for H3 and it is not necessarily true that classes with a large number of children will be more change-prone than other classes. We did find evidence of a range where this was the case, however.

4.3.5 AN EXPLANATION

One theory to explain this interval feature of inheritance found for DIT (and NOC) is that there is, at some point, a 'cognitive tipping point' effect in evidence. In other words, up until a certain level of complexity, change is relatively easy (this is the case where few descendents above need to be considered and DIT is low). Beyond that level, it becomes

difficult when both many ascendants below and many descendents above may need to be considered for any change. At some point deeper down, complexity starts to decline as the number of ascendants declines dramatically and the 'leaves' (i.e., those classes at the end of an inheritance hierarchy or tree) are reached (these classes have a high DIT). To summarize, we suggest that classes in the middle of an inheritance hierarchy may be perceived as *first class citizens* and need to be changed on a regular basis due to pressure from classes both *above and below* them. This pressure is not so acute on classes either side of the interval.

For NOC on the other hand, the demands placed on a super class by many immediate children may mean that the super class has to be changed in response to the ever-changing requirements of those subclasses. One might expect a class with many children to be changed relatively often. It is fairly counter-intuitive therefore to report that a class with > 20 children is less change-prone than classes with between 10 and 20 children. We suggest by way of explanation that with > 20 children, the developer becomes less inclined to make changes to the parent class than for classes with 10-20 children.

4.3.6 FAULT ANALYSIS

During the one year period over which faults were collected, 776 changes were made to classes to resolve identified faults in the WebCSC system. 495 of the 7,439 classes had fault fixes applied to them over the period studied. A large number of these classes (346) had only one fault fix applied during the period, followed by 90, 34 and 12 for two, three and four fixes being applied respectively. The highest number of fault fixes applied to a single class was 14. Table 4-5 shows the breakdown of number of changes made to resolve a fault per class. The mean number of faults per class across the whole system was 0.10.

No. faults	1	2	3	4	5	6	7	8	9	10	11	12	13	14
No. classes	346	90	34	12	6	2	0	0	1	1	0	1	1	1

TABLE 4-5 NUMBER OF FAULTS PER CLASS

4.3.7 HYPOTHESES H4-H6

In common with the analysis of changes, we explore the trends in faults for the WebCSC system through three further hypotheses.

Hypothesis H4: Are larger classes more fault-prone? A larger class has more functionality and there is a greater likelihood that some functionality in the class will need to be repaired as a result of a fault.

Hypothesis H5: Classes located high up in an inheritance hierarchy will be more fault-prone than other classes. Such a class has more dependents and there is therefore a greater likelihood that some functionality in the class will need to be enhanced and therefore be the cause of a fault.

Hypothesis H6: Classes with a large number of children are more fault-prone than other classes. This hypothesis is based on the belief that a class with many children will be the subject of greater maintenance activity, since there are added dependencies on the parent class because of the changing requirements of a large number of children.

4.3.8 CLASS SIZE AND FAULTS (H4)

Figure 4-11 to Figure 4-17 show the graphs of the correlations between class size features and faults. Table 4-6 shows the correlation coefficients for each of those figures. The Pearson correlation coefficient shows that all measures except for static methods were statistically significant when correlated against fault-proneness, while the Spearman rank correlation coefficients show all measures to be correlated significantly with faults, with the number of static methods being the weakest.

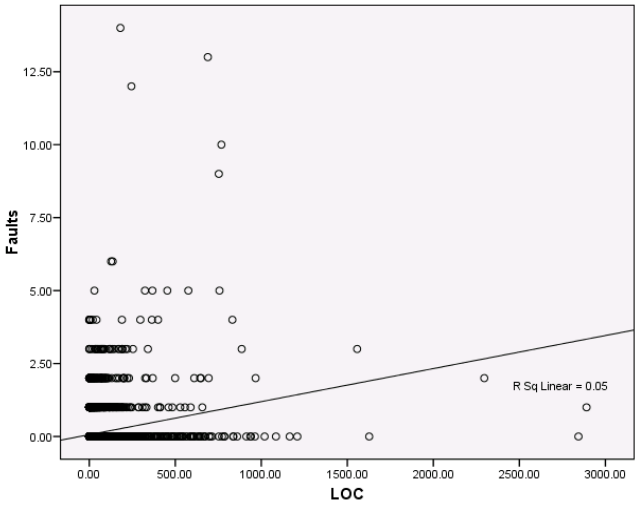

FIGURE 4-11 LOC VS NUMBER OF FAULTS

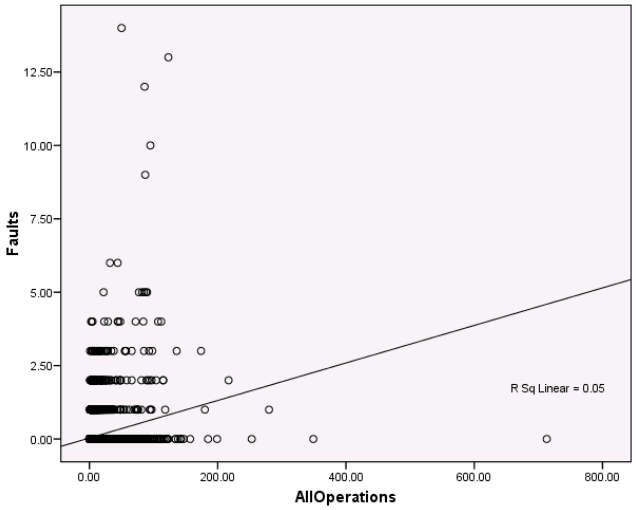

FIGURE 4-12 ALL OPERATIONS VS FAULTS

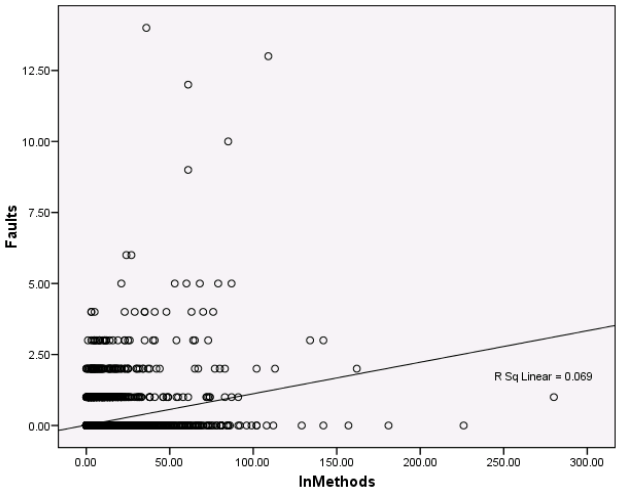

FIGURE 4-13 INSTANCE METHODS VS FAULTS

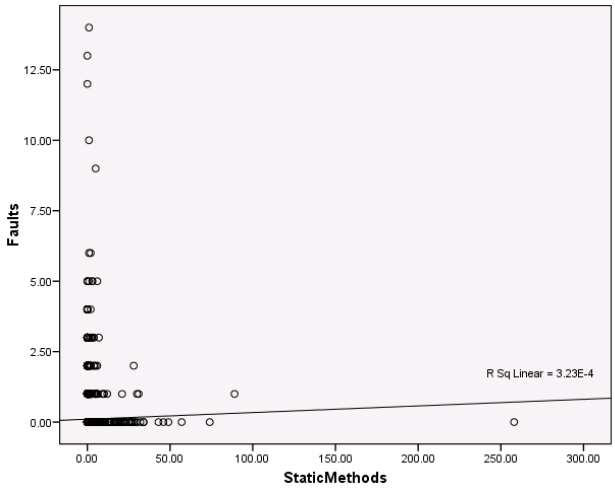

FIGURE 4-14 STATIC METHODS VS FAULTS

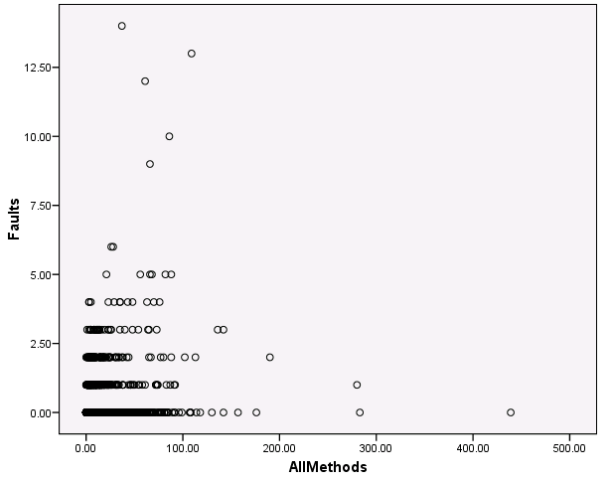

FIGURE 4-15 ALL METHODS VS FAULTS

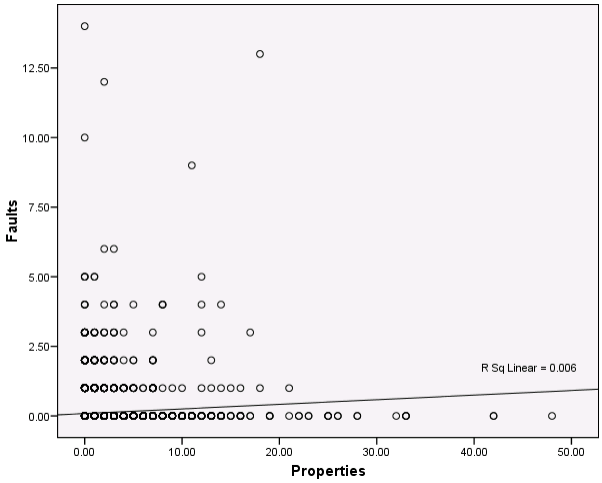

FIGURE 4-16 PROPERTIES VS FAULTS

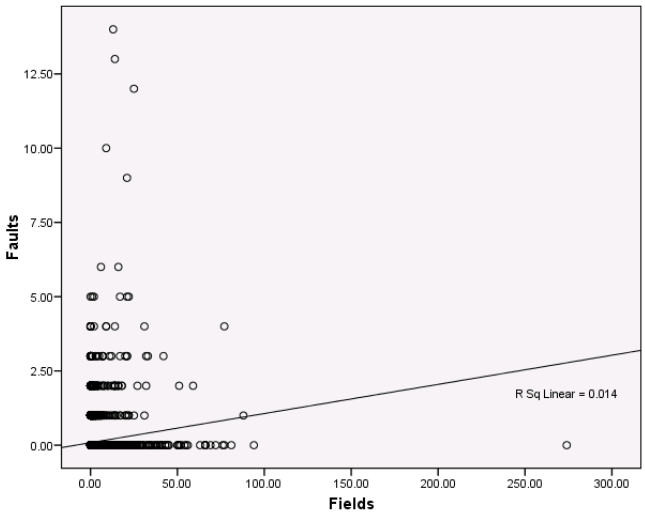

FIGURE 4-17 FIELDS VS FAULTS

Metric	Pearson's	Spearman's
LOC	0.22*	0.16*
Operations	0.22*	0.16*
Instance methods	0.26*	0.16*
Static methods	0.02	0.04*
Methods	0.24*	0.17*
Properties	0.08*	0.07*
Fields	0.12*	0.07*

TABLE 4-6 CORRELATION COEFFICIENTS (CLASS FEATURES VS FAULTS)

We can clearly find support for our original hypothesis H4 that large classes are more fault-prone. However, as we found for static methods (and interestingly the same trend as shown in Table 4-6), there are some class features that did not show as strong a correlation. In other words, the propensity of faults is not as great for static methods.

4.3.9 DIT ANALYSIS (H5)

To determine whether influence of inheritance characteristics on fault propensity we used the same two inheritance-based metrics as used previously, namely DIT and NOC.

Table 4-7 shows the number of classes and the mean number of faults per class grouped by DIT. Figure 4-18 clearly shows that once DIT exceeds 3, classes become more prone to faults; the evidence here again seems to support that trend. The interval effect is again clear between DIT 4 and 6 inclusive.

DIT	Number classes	Mean No. faults
0	1116	0.088
1	2445	0.090
2	1836	0.108
3	1362	0.065
4	296	0.193
5	92	0.272
6	41	0.585
7	214	0.192
8	52	0.058
9	31	0.645
10	1	0.000
11	1	0.000

TABLE 4-7 CLASSES AND MEAN NUMBER OF FAULTS PER CLASS GROUPED BY DIT

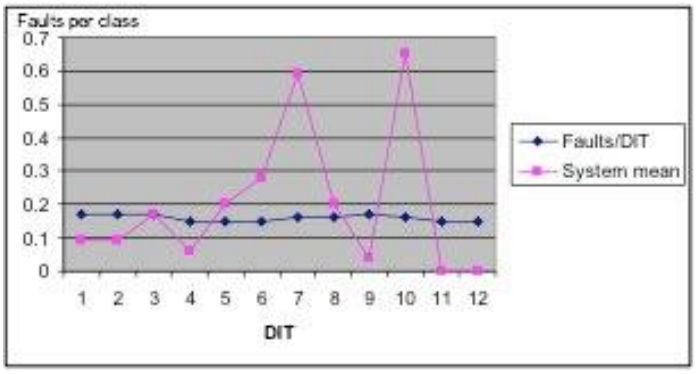

FIGURE 4-18 MEAN FAULTS PER DIT

We find little support for hypothesis H5 from the available data. It is not true that classes with a high DIT are more fault-prone than other classes. In keeping with the results so far for change-proneness, it is in the middle tiers of the inheritance hierarchy where faults tend to occur.

One noteworthy observation for the analysis so far is therefore that classes at deep levels tend to be the most change-prone and fault-prone. A recent study by Nasseri et al., (Nasseri, Counsell, & Shepperd, 2008) reported that 96% of incremental class changes over the course of the versions of four Java open-source systems studied were at inheritance levels 1 and 2 (where level 1 is immediately below Object). Only 4% of changes were made at levels 3 and below; this was largely because the majority of the system's classes were at DIT 1 and 2. It would appear that maintaining shallow DIT levels might be one policy that developers adopt to avoid the problems that we see emerging for the WebCSC system (i.e., many changes and faults at middle tiers of the inheritance hierarchy). That is not to say, of course, that there is conclusive proof that a shallow inheritance hierarchy is any better in terms of fault propensity. For open-source systems however, it seems to be a common policy to follow. The nature of open-source with geographically disparate developers, who may not be aware of the overall system design, may be the root cause of very shallow inheritance hierarchies. If a developer is not familiar with the overall inheritance hierarchy, then that might inhibit certain changes being made to the same hierarchy by that developer.

4.3.10 NOC ANALYSIS (H6)

The NOC values collected showed that the majority of classes had zero subclasses. Those classes had a similar rate of faults to the mean of the entire system (0.10 faults

per class). Classes with an NOC between 1 and 10 however, had a higher rate of faults than the mean of the system. This increased for classes with an NOC between 11 and 20 and again for classes with an NOC above 20. The NOC measurements clearly show a trend for classes with a higher number of children to have an increased propensity for faults. Table 4-8 shows the number of classes and the mean number of faults per class grouped by the number of children. The striking feature of Table 4.8 is the jump from NOC 1-10 to NOC 11-20 (the mean faults almost double in this transition). It is interesting that there is no interval effect in evidence as there is for inheritance depth or for NOC changes.

NOC	Classes	Mean faults
0	5951	0.10
1-10	1412	0.13
11-20	75	0.23
>20	49	0.25

TABLE 4-8 NOC, NUMBER OF CLASSES AND MEAN NUMBER OF FAULTS

In terms of our original hypotheses H6, we conclude that classes with a large number of children are more fault-prone than those classes in a simpler inheritance hierarchy. Since the values in Table 4.8 show that classes with greater numbers of children were more likely to be changed, it implies that the number of faults in those classes are likely to be correspondingly fault-prone. One lesson that we can learn from our analysis is that restricting the number of children *per se* can go some way to limiting the number of changes likely to be made to a class and potentially the number of faults.

4.3.11 HYPOTHESES H7-H8

In common with the analysis of inheritance and size, we explore the trends in coupling, cohesion and their relationship with faults for the WebCSC system through two hypotheses.

Hypothesis H7: Are classes with higher coupling more fault-prone? A class with higher coupling has more dependencies and there is a greater likelihood that some functionality in the class will need to be repaired as a result of a fault.

Hypothesis H8: Are less cohesive classes more fault-prone? A less cohesive class is likely to be less reusable and less comprehendable and, as such, there is a greater likelihood that some functionality in the class will need to be repaired as a result of a fault.

4.3.12 COUPLING AND FAULT-PRONENESS (H7)

We measured two coupling metrics: afferent coupling (Ca) and efferent coupling (Ce). Classes with lower coupling are said to be preferred as they tend to exhibit more traits of maintainable code, such as readability and reuse. The mean value of afferent coupling for all classes in the system was 5.482. The subset of the classes in the system that had faults during the period studied, however, had a significantly higher mean value of afferent coupling of 8.047. Similarly, the mean value of efferent coupling for the subset of classes in the system that had faults (26.813) was also significantly higher than the mean value for the whole system (19.351). The median values of afferent coupling are the same for all classes, classes with faults and classes without faults; for efferent coupling the group of classes with faults had a higher median value (20) than the group of all classes (14) and the group of classes without faults (13).

Table 4-9 shows the afferent coupling and efferent coupling values; the faulty classes in general have higher coupling.

	Mean Value	Median Value
Ca for all classes	5.482	2
Ca for classes with faults	8.047	2
Ca for classes without faults	5.119	2
Ce for all classes	19.351	14
Ce for classes with faults	26.813	20
Ce for classes without faults	18.296	13

TABLE 4-9 COUPLING METRICS

Figure 4-19 shows a box plot graph with the comparative afferent coupling values for classes with faults and classes without faults. While the upper quartile and median are marginally higher for the faulty classes, the higher extremes and outliers have much higher coupling for the faultly classes than the non-faulty classes. This compares with Figure 4-20, showing the box plot graph of efferent coupling values for classes with faults and classes without faults. In this case, the higher extremes and outliers are similar for faulty and non-faulty classes, and it is the lower quartile, upper quartile, and median that have higher coupling for faulty classes.

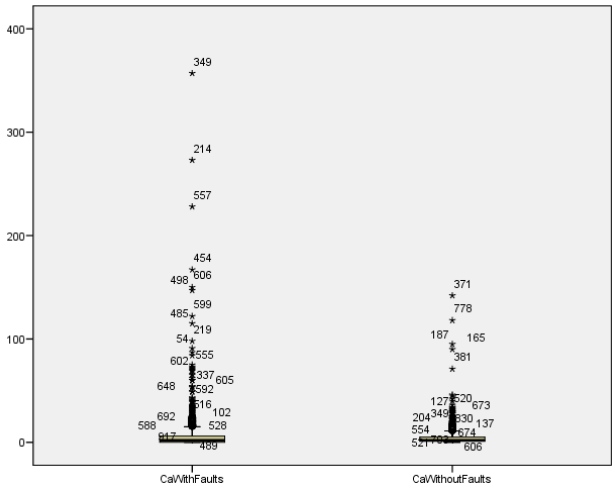

FIGURE 4-19 CA PER FAULTY AND NON-FAULTY CLASS

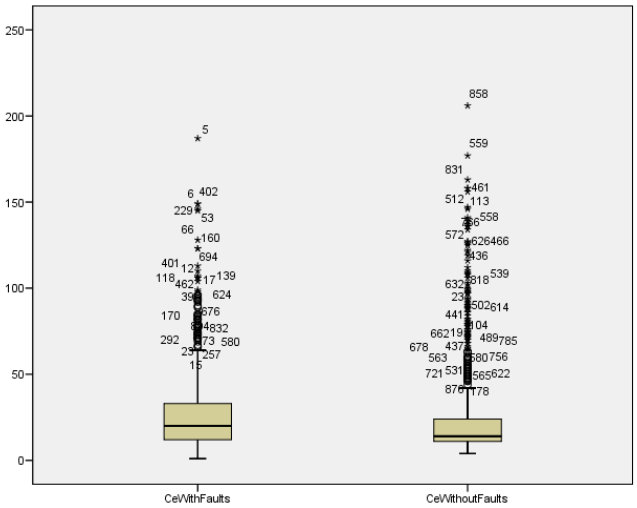

FIGURE 4-20 CE PER FAULTY AND NON-FAULTY CLASS

As the box plots in Figure 4-19 and Figure 4-20 are difficult to visually inspect we perform further analysis using the Mann-Whitney U test. The Mann-Whitney U test is a non-parametric test for assessing whether two independent samples have equally large values and compares the median values of the two samples. First, we applied the Mann-Whitney U test to two samples: the set of classes without faults and the set of classes with faults and we tested the efferent coupling values. The Mann-Whitney U test showed the difference between the two samples as highly significant ($P<0.001$, two tailed test) where n1 = 6560, n2 = 925 and U = 3959463.5, confirming our findings from our inspection of the boxplot, that the sample of classes with faults tended to have a higher efferent coupling value than the sample of classes without faults. Second, we

applied the Mann-Whitney U test across the same two samples and we tested the afferent coupling values. The Mann-Whitney U test showed the difference between the two samples as highly significant (P<0.001, two tailed test) where n1 = 6560, n2 = 925 and U = 3270905.0, showing that the sample of classes with faults tended to have a higher afferent coupling value than the sample of classes without faults.

Finally, we correlated the coupling values with the number of faults a class had during the period studied, and also the net number of lines of code changed (that is, the sum of LOC added, modified or deleted) to rectify a fault in the same period.

Table 4-10 Correlation Coefficients shows the correlation coefficients using Spearman and Pearson correlations. We found a correlation between the afferent coupling and the number of faults a class had experienced, significant at the 1% level using both Spearman's (0.046) and Pearson's (0.035) correlations. While we found a correlation between afferent coupling and the net number of lines of code changed to rectify faults (0.029), significant at the 5% level, using Spearman's correlation; Pearson's correlation did not show a significant correlation. When we calculated the same correlations for efferent coupling, we found stronger correlations: 0.181 and 0.216 (Spearman's and Pearson's, respectively) for the correlation between efferent coupling and the number of faults in a class, significant at the 1% level, and 0.148 and 0.142 (Spearman and Pearson, respectively) for the correlation between efferent coupling and the number of lines of code changed (added, modified and deleted) to rectify a fault, again, significant at the 1% level.

Correlation	**Spearman**	**Pearson**
Ca and number of faults	0.046*	0.035*
Ca and LOC changed to rectify faults	0.029**	0.06
Ce and number of	0.181*	0.216*

faults		
Ce and LOC changed to rectify faults	0.148*	0.142*

* Significant at the 1% level

** Significant at the 5% level

TABLE 4-10 CORRELATION COEFFICIENTS

In summary, we identified a relationship between the fault propensity of a class and it's coupling. We have showed that the mean value for efferent coupling and afferent coupling are higher for classes that had faults during the 24 month period studied, than for classes that do not have faults. We have shown there is a significant correlation between afferent coupling and efferent coupling of a class and the number of faults, and we have also shown a significant correlation between the efferent coupling and the number of lines of code changed (added, modified and deleted) to rectify faults during the period studied. Interestingly, we found that efferent coupling has more of a bearing on fault-proneness than afferent coupling. In terms of our original hypotheses H7, we conclude that a class with high coupling tends to be more fault-prone.

4.3.13 COHESION AND FAULT-PRONENESS (H8)

Cohesion measures how strongly related the functionality expressed in a class is. A class that is more cohesive is said to be preferrable for similar reasons to low coupling: as they tend to exhibit traits of maintainable code such as readability, reuse and robustness. We measured cohesion using the Lack of Cohesion in Methods (Henderson-Sellers) metric (LCOM(HS)). The mean LCOM(HS) value for all classes in the system was 0.702. Taking the set of classes that contained faults during the period studied gave an LCOM(HS) value of 0.691, while the set of classes that contained no faults during the period studied gave a marginally higher LCOM(HS) value of 0.704. The median values for the three groups of classes are identical.

Table 4-11 shows the full set of LCOM(HS) values for all sets of classes measured. The LCOM(HS) values for the three sets of classes are very close; this would suggest that the

cohesiveness of the classes (measured using LCOM(HS)) does not impact upon the fault-proneness of the classes.

	Mean	Median
LCOM(HS) for all classes	0.702	0.78
LCOM(HS) for classes with faults	0.691	0.78
LCOM(HS) for classes without faults	0.704	0.78

TABLE 4-11 COHESION VALUES

Figure 4-21 shows a box plot for the LCOM(HS) values for the group of classes containing faults and the group of classes containing no faults. The distribution is similar: the median values are equal and the lower and upper quartiles are in a similar range. The higher and lower extremes and outliers are greater for the group of classes with no faults.

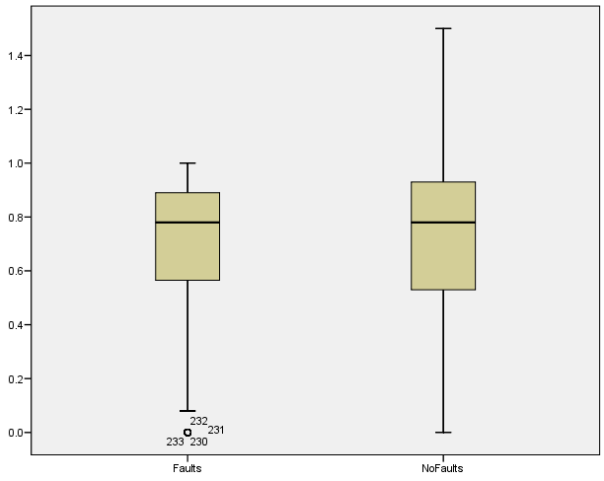

FIGURE 4-21 LCOM(HS) VALUES FOR FAULTY AND NON-FAULTY CLASSES

We correlated the LCOM(HS) value of a class with the number of faults the class experienced during the period studied using both the Spearman and Pearson correlations. Table 4-12 shows the resulting correlation coefficients. The Spearman correlation gave a negative correlation, showing that the lower the cohesiveness of the class the more likely it is to contain faults. The Pearson correlation, however, contradicted this, showing that the higher the cohesiveness of the class, the more likely it is to contain faults. Both Spearman and Pearson correlations were weak, but significant at the 5% level.

Correlation	**Spearman**	**Pearson**
LCOM(HS) and faults	-0.033**	0.034**

** significant at the 5% level

TABLE 4-12 CORRELATION COEFFICIENTS

In summary, we did not find a strong relationship between the cohesiveness of a class and fault-proneness, when cohesiveness if measured by LCOM(HS). In terms of our original hypotheses H8, we cannot conclude that a class with low cohesion tends to be more fault-prone.

4.4 DISCUSSION

The study presented raises a number of issues for the developer and a strategy for minimizing a) the likely changes that need to be made to a class and b) the faults that arise from a class.

First, the argument in favour of limiting the depth to which an inheritance hierarchy should grow results in a dilemma - restricting inheritance depth will inevitably cause inheritance width to grow to compensate – and we have shown how in terms of NOC that can also be problematic. It would seem that, from our study, the size of the entire inheritance hierarchy (both in depth and width) should perhaps be restricted so that depth and width are moderated. Our data suggests DIT 3 to be the threshold value and NOC to be limited to as small as possible a value. Second, the study highlights a) the need to maintain a pragmatic view of the entire hierarchy and b) the vigilance needed on the part of developers and project managers to apply consistent, remedial

techniques such as refactoring (Fowler M. , 1999) and re-engineering; for example, the replacement of inheritance with aggregation or other forms of coupling (Johnson & Opdyke, 1993). It is ironic that the least applied refactorings identified in the previous chapter (i.e., inheritance-based refactorings) are those which, if undertaken, may strike at the heart of change and fault-proneness in a system. Third, if, as we suggest, there is a level of DIT and NOC which show a higher propensity for faults, then what is a project manager or developer to do in the face of consistent pressure for a system to grow in size as it evolves (Girba, Lanza, & Ducasse, 2005)? We could propose that a code smell analysis (Fowler M. , 1999) could be used to determine the point at which re-engineering and/or refactoring should take place and, in that sense, the warning signs can be highlighted. Equally, there may be a case for amalgamating classes or even collapsing a hierarchy to avoid its depth becoming too large.

For a study of this type, the threats to its validity also need to be considered (Fenton & Pfleeger, 1997). First, we have to consider that only one system was used as a basis of the study. However, we feel we are adding to the knowledge already accumulated by the two previous studies of Bieman et al., and, in that sense, our work is a contribution. Second, we have used a C# system and previous studies have used a combination of C++ and Java only. In defense of this threat, the differences between C# and Java are relatively minimal. We feel that our study actually adds to our knowledge of trends in different object-oriented languages. Third, since we have collected faults and change data, one criticism is that they are likely to produce the same results anyway since most faults induce the requirement to make code changes. However, the period in which we studied the faults for the WebCSC system was for a shorter period than that for changes. Also, the fault data represented only a very small part of the overall changes made to the system. Finally, we have assumed that each 'change' and 'fault' are equivalent in nature, when in actuality there would be large differences in the size of a) each change made and b) severity of each fault fixed. Future work could consider a form or normalization for change size and/or fault to determine if any different results became evident.

4.5 CONCLUSIONS

In this chapter we have described an empirical study into design context and change and fault-proneness, specifically: inheritance, coupling, cohesion and size.

The objectives of this chapter were to:

1. Identify if the key aspects of an object-oriented design context: inheritance, coupling and cohesion, and size were contributory factors of fault and change propensity.
2. Identify if there were properties or characteristics of these aspects of the design context that cause higher fault or change-proneness.

Change was measured against the design context of the classes within the system, more specifically size, inheritance characteristics and coupling and cohesion properties. Results showed a strong positive correlation between the class size measures and change-proneness but this was not true for class features studied. Classes within a specific range of inheritance depth and number of children were found to be relatively more prone to change - the fault data showed similar results. The most striking result to emerge was the notion of an inheritance depth 'interval' between which change and fault-proneness were at their highest. Below and above that interval, however both features were less acute. We found that coupling had a significant correlation with the fault-proneness of a class. Efferent coupling was a stronger indicator of fault-proneness than afferent coupling. Cohesion, when measured using the LCOM(HS) metric, however, did not show a strong relationship with the fault-proneness of the classes in the system studied.

One question that arises from this study is how the results could inform developer practice. The first issue is that if large classes are changed more often, then would decomposing large classes through, for example, refactoring actually reduce the total number of changes? One view would be that relatively small classes are more cohesive and hence while this might not mean a reduction in the number of changes, any necessary changes to smaller classes are likely to be more efficiently achieved. In other words, active re-engineering might pay dividends at a later stage. Finally, from an inheritance perspective, it would seem that extending the hierarchy beyond a certain level might be the cause of significant extra maintenance activity. Most evidence suggests very shallow inheritance hierarchies exist in object-oriented systems and the evidence presented in this chapter suggests that, counter-intuitively, restricting inheritance depth might be a sensible strategy.

5 CHAPTER 5 – THE RELATIONSHIP BETWEEN DESIGN PATTERNS AND FAULT AND CHANGE-PRONENESS

5.1 INTRODUCTION

In the previous chapter we documented an empirical investigation into the design context of a class (in terms of inheritance characteristics, coupling, cohesion and size) and change and fault-proneness. Our results showed a correlation between some aspects of the design context and change and fault-proneness. In this chapter we extend this empirical study to consider if design pattern participation has a similar relationship to fault and change-proneness. The chapter represents an extension to the study in Chapter 4 and a further replication of a previous study by Bieman et al. It uses the same change and fault data used in the previous chapter.

Our research objectives for this chapter are:

1. To understand how design patterns are applied in commercial software.
2. To determine which patterns are most common and how often are they applied.
3. To determine whether design pattern participation results in less change-prone and fault-prone classes.

Design patterns are reusable descriptions or templates showing the relationships and interactions between classes and objects. Many design patterns including some of those described by Gamma et al., (Gamma, 1995), promote adaptability by supporting specialization of the pattern-based classes. A system built using design patterns can therefore be adapted by creating concrete classes with desired new functionality rather than by direct modification of the existing set of core pattern-based classes. It is therefore reasonable to expect that design pattern 'participant' classes (i.e., those core classes) would have a relatively lower propensity for change relative to all other classes in a system since, in theory, they should remain untouched by developers. A previous study by Bieman et al., (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003) showed that the opposite was true; design pattern participants were actually more change-prone than non-pattern classes. The purpose of this chapter is to explore change and fault data to support or refute the conclusions of that earlier study.

As design patterns are often at the core of software systems this observation is of particular interest when studying the evolution and future maintenance requirements of a software system. As previous studies have shown, there is a correlation between a class' propensity for change and the same class' fault-proneness. It might be expected that design patterns are not only more change-prone, but consequently more fault-prone.

In our empirical investigation, we replicate and extend Bieman's study (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003) to explore whether design pattern participants are indeed more change-prone in the studied system and we extend Bieman's study to determine whether pattern classes are more fault-prone than non-pattern classes (the study of Bieman et al., did not use fault data).

For clarity, for the remainder of this chapter we define a class as belonging to one of two mutually exclusive sets: a pattern-based class participates in an implemented design pattern; a non-pattern class denotes a class with no implemented relationship with a documented design pattern, although it may be connected (i.e., coupled) to classes that are.

The remainder of the chapter is organised as follows. In the following section, we describe the motivation for the study. In Section 5.2, we describe preliminaries such as the system studied and the metrics extracted by the tool. We then analyse the data, discuss the key issues raised by the study (Section 5.3) and suggest threats to the validity of the study; finally, we conclude and point to further work in Section 5.4.

It should be noted that the basis of this research was published in (Gatrell, Counsell, & Hall, 2009) and (Gatrell & Counsell, 2011).

5.2 METHOD

In common with Bieman's original study (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003) and to remain faithful to that study, we focused our investigation on intentional design patterns (i.e., a design pattern was implemented with the explicit intention to follow a documented design pattern). The manual approach used to identifying design patterns (also in keeping with Bieman's study) consisted of four steps:

1. Search for design pattern names in the documentation.
2. Identify the context of the classes identified in Step 1 by analyzing the object models (both the original design artefacts, such as UML diagrams, and models generated directly from the source code). Once the classes whose documentation specified a class relating to a pattern/role were found, the object models were inspected to identify the classes constituting that pattern. Links between classes that implemented the pattern were then searched for.
3. Verify that the candidate pattern was a pattern instance and examine the pattern implementation to look for lower level details, for example, required delegation constructs.
4. Verify the purpose of the pattern. Each group of classes representing a pattern candidate was examined to confirm that the classes and relations had the same purpose as described by Gamma et al., (Gamma, 1995).

5.2.1 DESIGN PATTERN DETECTION

The same design patterns as those in Bieman et al.'s were included as part of the study which we now describe in the following sub-sections.

5.2.1.1 THE ADAPTOR DESIGN PATTERN

The adaptor design pattern translates one interface for a class into a different interface allowing classes to work together that normally could not due to incompatible interfaces.

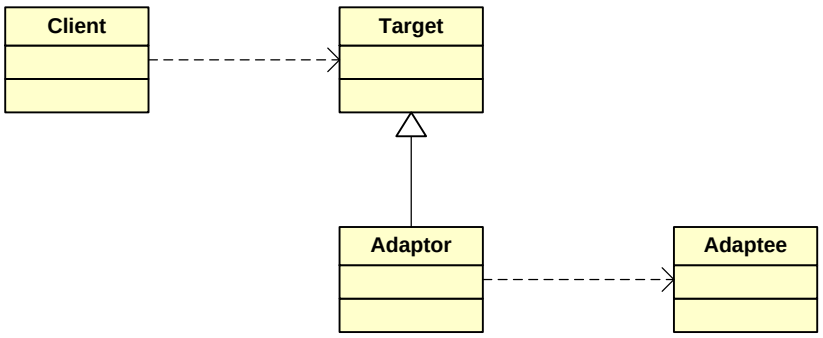

FIGURE 5-1 THE ADAPTOR DESIGN PATTERN

Figure 5-1 shows the class structure of the adaptor design pattern. The **Client** class calls a method on the type **Target** which implements an interface that the **Client** is

expecting. The **Adaptor** class inherits from the **Target** type and translates the call from the **Client** into a call that the **Adaptee** accepts, allowing the **Client** and **Adaptee** to communicate.

5.2.1.2 THE BUILDER DESIGN PATTERN

The builder design pattern separates object construction from its representation, by abstracting the steps of construction. Different implementations of these steps can be provided resulting in different representations of the object.

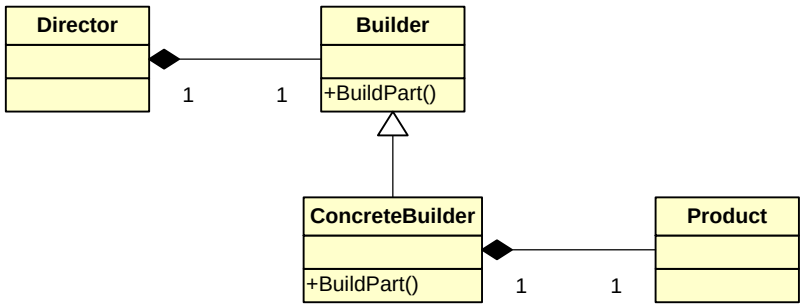

FIGURE 5-2 THE BUILDER DESIGN PATTERN

Figure 5-2 shows the class structure for a builder design pattern. The **Director** class calls **BuildPart** on a collection of **Builder** instances. The **ConcreteBuilder** is responsible for constructing and initializing the different **Products** required.

5.2.1.3 THE COMMAND DESIGN PATTERN

The command design pattern encapsulates a command request as an object, recording all the information needed to call a method at a later time, such as the method name, the object that owns the method and any parameter values.

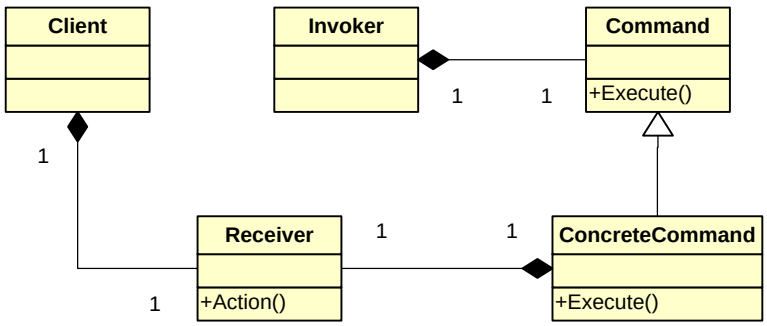

FIGURE 5-3 THE COMMAND DESIGN PATTERN

Figure 5-3 shows the class structure for a command design pattern. The **Client** calls **Execute** on a **ConcreteCommand** instance that has been added as a **Command** for the *client.*

5.2.1.4 THE CREATOR DESIGN PATTERN

The creator design pattern (Factory Method) defines classes with responsibility for creating new instances of other classes, based on whether the original class contains, aggregates or contains initialising information about the class being instantiated.

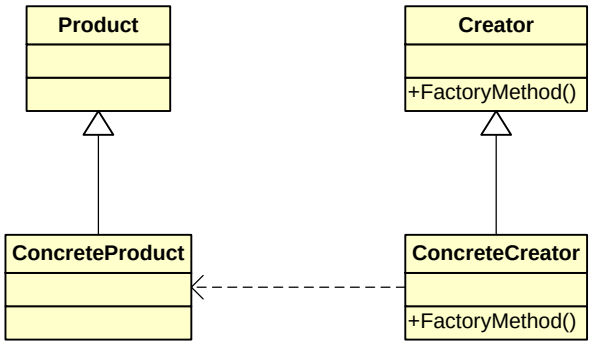

FIGURE 5-4 THE CREATOR DESIGN PATTERN

Figure 5-4 shows the class structure for the creator design pattern. The **ConcreteCreator** takes the responsibility for creating new concrete instances of **Products** away from any calling classes.

5.2.1.5 THE FACTORY DESIGN PATTERN

The factory design pattern (abstract factory) creates an instance of several derived classes, by defining a method for creating objects that can be overridden by subclasses to specify the derived type.

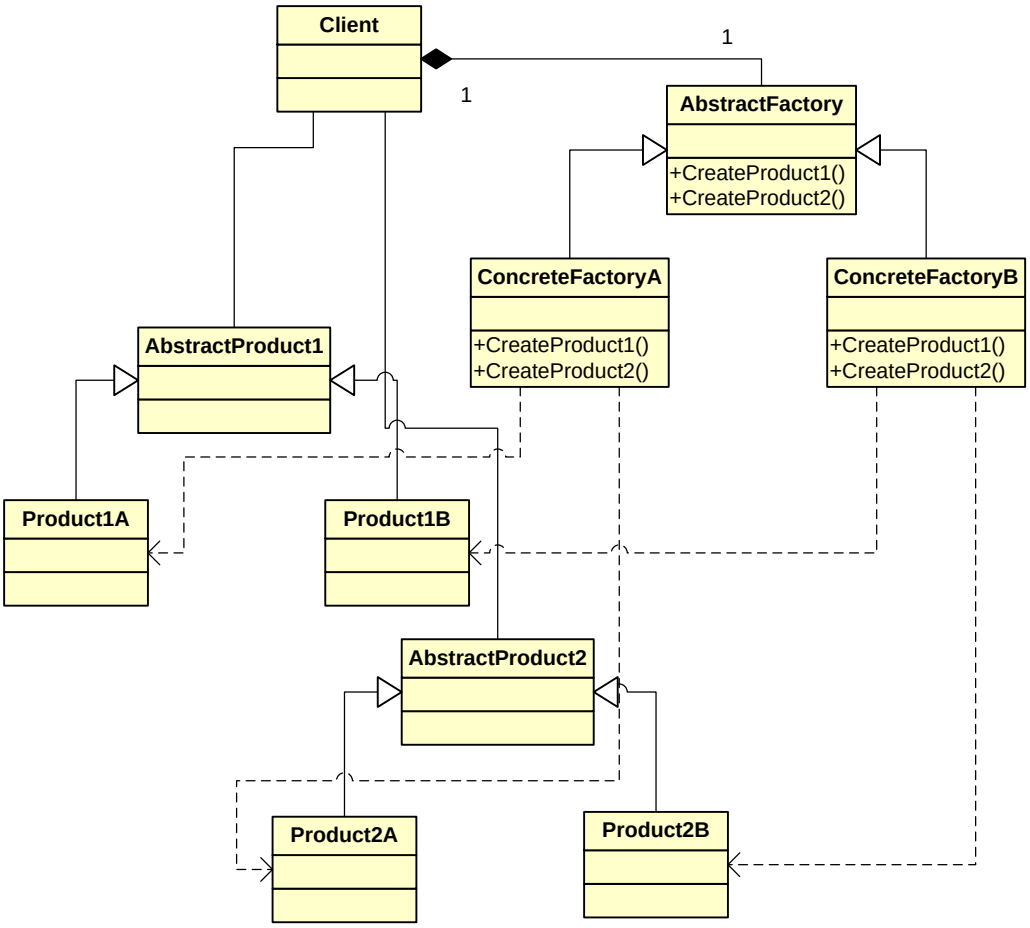

FIGURE 5-5 THE FACTORY DESIGN PATTERN

Figure 5-5 shows the class structure for a factory design pattern. The **ConcreteFactoryA** object takes responsibility for creating instances of **Product1A** and *Product2A* classes, and the **ConcreteFactoryB** object takes responsibility for creating instances of **Product1B** and **Product2B** classes.

5.2.1.6 THE TEMPLATE METHOD DESIGN PATTERN

The template method design pattern allows subclasses to implement behaviour that can vary by method overriding.

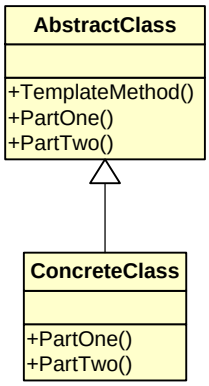

FIGURE 5-6 THE TEMPLATE METHOD DESIGN PATTERN

Figure 5-6 shows the class structure for the template method design pattern. The **AbstractClass** object defines a **TemplateMethod** that calls two methods (**PartOne** and **PartTwo**) that can be overridden by the **ConcreteClass**. This allows each **ConcreteClass** to provide its own implementation for part or all of the functionality.

5.2.1.7 THE FILTER DESIGN PATTERN

The filter design pattern allows pluggable filters to intercept incoming or outgoing messages or lists, filters can be added or removed unobtrusively without modifying existing code.

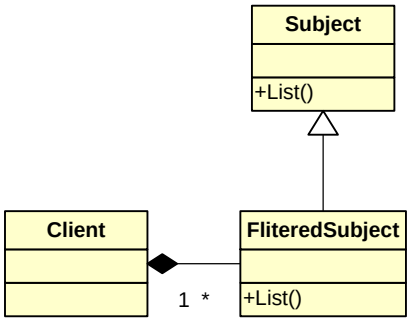

FIGURE 5-7 THE FILTER DESIGN PATTERN

Figure 5-7 shows the class structure for the filter design pattern. The **Client** calls the overridden accessor method on the **FilteredSubject** which inherits from the **Subject** but filters the results from the **Subject** before the **Client** receives the result.

5.2.1.8 THE ITERATOR DESIGN PATTERN

The iterator design pattern sequentially accesses elements of a collection without exposing the underlying representation.

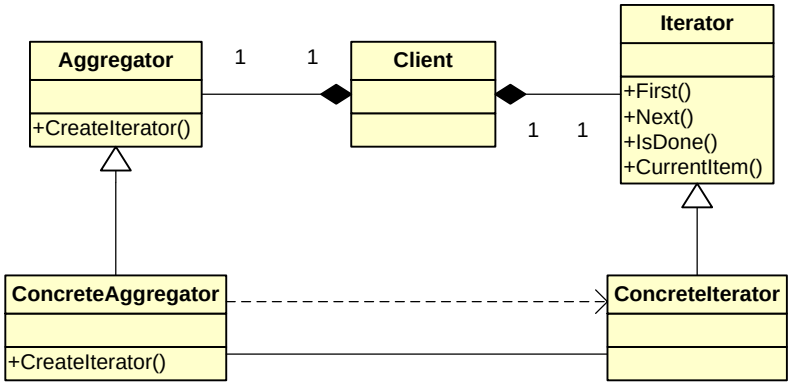

FIGURE 5-8 THE ITERATOR DESIGN PATTERN

Figure 5-8 shows the class structure for the iterator design pattern. The **Iterator** objects allow access to, and navigation around, the **Aggregator** class structure without need for knowledge of the **Aggregators** underlying representation.

5.2.1.9 THE PROXY DESIGN PATTERN

The proxy design pattern makes an object represent another object by allowing a class to function as interface to a different object.

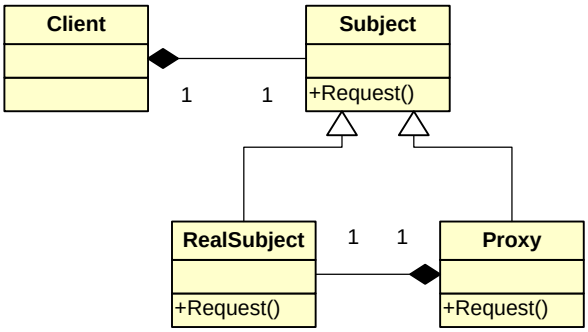

FIGURE 5-9 THE PROXY DESIGN PATTERN

Figure 5-9 shows the class structure for the proxy design pattern. The **Proxy** object is presented to the **Client** and marshalls calls through to the **RealSubject**.

5.2.1.10 THE SINGLETON DESIGN PATTERN

The singleton design pattern is a class of which only one instance can exist by restricting the instantiation to one object.

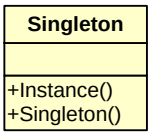

FIGURE 5-10 THE SINGLETON DESIGN PATTERN

Figure 5-10 shows a singleton class. The **Singleton** constructor ensures only one instance of the class is created. New clients calling the constructor are passed the existing and sole instance of the class instead of instantiating a new one.

5.2.1.11 THE STATE DESIGN PATTERN

The state design pattern allows the alteration of an object's behaviour when its state changes.

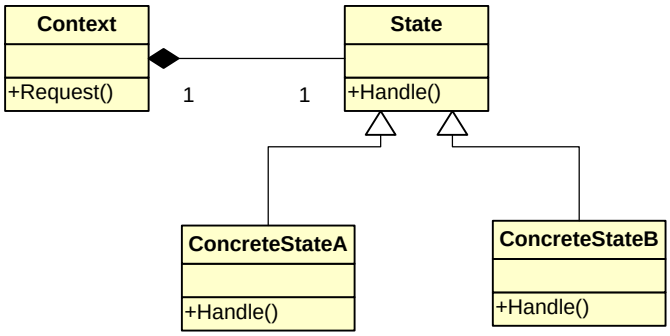

FIGURE 5-11 THE STATE DESIGN PATTERN

Figure 5-11 shows the class structure for the state design pattern. Each state is represented by a different concrete class. Calling **Handle** on the state will provide different behaviour depending on which **State** (i.e., which concrete state class) the **Context** has.

5.2.1.12 THE STRATEGY DESIGN PATTERN

The strategy design pattern encapsulates an algorithm within a class allowing algorithms to be swapped dynamically.

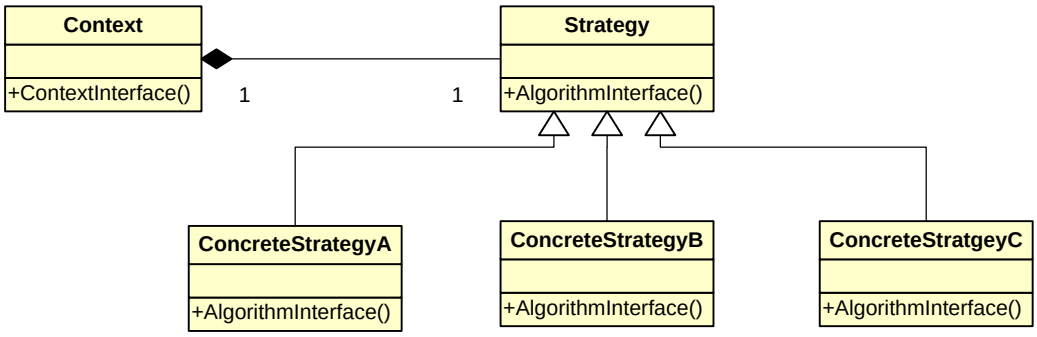

FIGURE 5-12 THE STRATEGY DESIGN PATTERN

Figure 5-12 shows the class structure for the strategy design pattern. Different implementations of the **Strategy** are represented by the different concrete classes.

5.2.1.13 THE VISITOR DESIGN PATTERN

The visitor design pattern defines a new operation to a class without change by separating the algorithm from the object structure on which it operates.

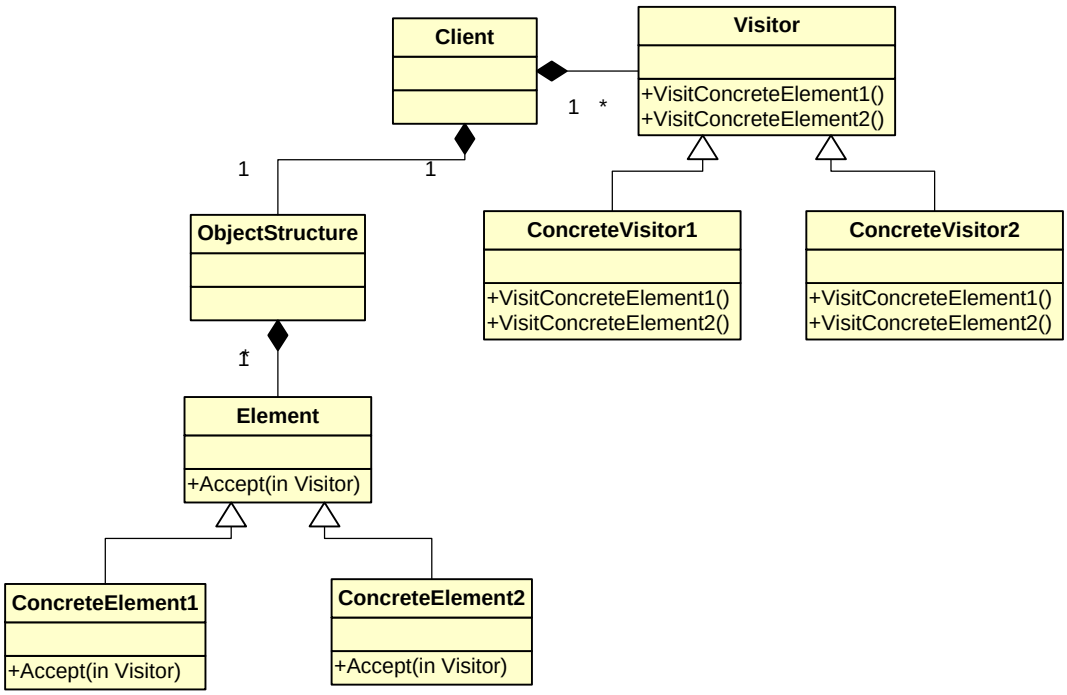

FIGURE 5-13 THE VISITOR DESIGN PATTERN

Figure 5-13 shows the class structure for the visitor design pattern. A feature can be implemented in a concrete **Visitor** class and be applied to an object structure by a concrete **Element** accepting and giving access to that visitor, via the **Accept** method.

5.2.2 CHANGE AND FAULT PROPENSITY

The change and fault propensity for each class was collected from the fault history in the source control system over a period of two years. As part of the enforcement policy of the source control system, a check-in policy was created to force the developer submitting a code change to the source control server to manually associate the source code modification with a development task (either a change request or a fault from the fault tracking system). Because of this we were able to identify modifications made to the source code specifically to implement a change or to rectify a fault. For each change made to a class to rectify a fault, the lines of code (LOC) that were added, deleted and modified were collected. An added line was recorded every time a new line of code was inserted in order to correct a fault; similarly for deleted and modified lines of code. We define a LOC as an executable line of code. Comments as well as method, class and namespace declarations were not therefore considered in this category.

5.3 DATA ANALYSIS

In this section, we explore the results of the empirical investigation. First, we look at the design patterns that were detected in the system; second, we look at the relationship between design pattern participation and change; finally, we look at the relationship between design pattern participation and faults.

5.3.1 DETECTED DESIGN PATTERNS

In total, 658 out of the 7439 classes were found to participate in design patterns. Classes related to the Singleton and Strategy patterns were found to be the most popular (with 168 and 101 classes, respectively), while Creator and Method-based classes were less common with only 5 and 8 participant classes, respectively. No Iterator-based classes were found and only one Proxy was identified in the system. Table 5-1 shows the number of design patterns detected in full.

Pattern	Classes
Adaptor	74
Builder	64
Command	26
Creator	5
Factory	97
Method	8
Filter	16
Iterator	0
Proxy	1
Singleton	168
State	32
Strategy	101
Visitor	66
Total	658

TABLE 5-1 DETECTED DESIGN PATTERNS

5.3.2 PATTERN CLASSES AND CHANGE

The mean number of changes for classes that did not participate in intentional design patterns was 2.51 changes per class, contrasted with the mean number of changes for classes that *did* participate in intentional design patterns of 6.34 per class. Five patterns, namely Adaptor, Template Method, Proxy, Singleton and State were found to have a very high change rate compared with the system as a whole. The Strategy and Visitor patterns were also found to have a relatively higher propensity to change. The Builder, Factory and Filter were found to have a near normal propensity for change (when compared to the system mean), while Command and Creator patterns actually had a

lower propensity for change than the average for the whole system. Table 5-2 shows the change data for each of the 13 patterns.

When manually inspecting a sample of the pattern classes that had been subject to higher than normal rate of change, we noticed that many of these classes contained implementation code that should not have been the responsibility of the pattern class to provide. We suspect that because of the strong familiarity with pattern classes, system developers naturally chose these pattern classes as the focal point for making changes, instead of immediately surrounding classes (which would have been a more rational course of action). This was particularly the case for the highest change-prone patterns: Adaptor, Method, Proxy, Singleton and State. In other words, our study sheds some light on why pattern-based classes may be more change-prone – it is simply due to familiarity with those classes to developers which make them easy to adapt and change.

Pattern	Classes	Changes	Average
Adaptor	74	466	6.38
Builder	64	176	2.75
Command	26	37	1.42
Creator	5	6	1.20
Factory	97	265	2.73
Method	8	64	8.00
Filter	16	39	2.44
Iterator	0	0	0.00
Proxy	1	6	6.00
Singleton	168	822	4.89
State	32	146	4.56
Strategy	101	371	3.67
Visitor	66	267	4.05
Total	658	2665	

TABLE 5-2 CHANGES FOR PATTERN CLASSES

Figure 5-14 shows the box plot for participant classes in the studied design patterns. It confirms that classes participating in patterns were more prone to change than classes that do not. The lower quartile, upper quartile and median are higher for pattern-based classes. The upper extremes and highest outlier however are higher for non-pattern classes.

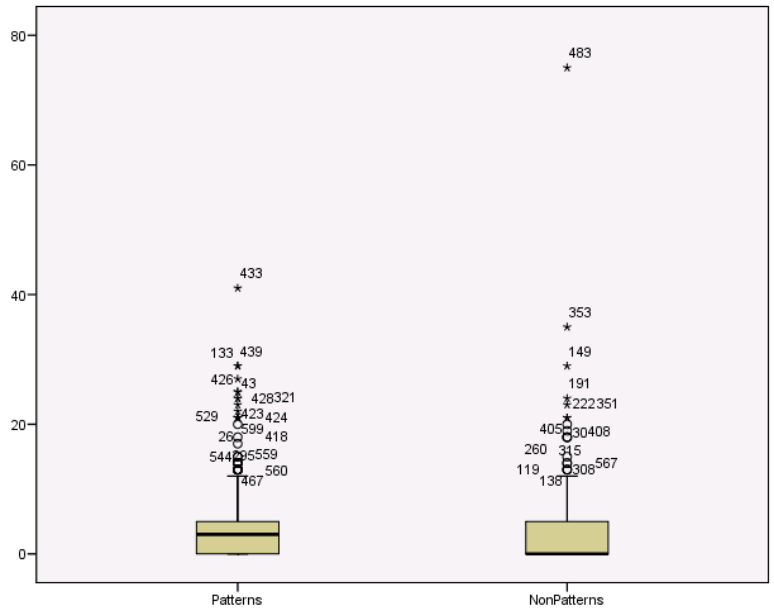

FIGURE 5-14 NUMBER OF CHANGES PER CLASS

One question that naturally arises from this analysis is why certain patterns were changed more often than others? The answer is relatively straightforward in the case of the WebCSC system. The patterns changed most frequently were those that comprised the core design of the system. Pattern classes that played more of an ancillary role did not tend to be changed as frequently. In other words, it was the role of the pattern in the system rather than the pattern *per se* that produced this result.

The result clearly supports Bieman's earlier finding with respect to pattern classes. It would seem that although patterns are being used extensively, their contextual use is being perverted by 'improper' developer practice of frequent change. This practice could be dangerous for the future structure and maintainability of the system. We see clear support for the claim and the earlier result of (Bieman, Straw, Wang, Willard Munger, & Alexander, 2003; Bishop, 2008; Buschmann, Meunier, Rohnert, Sommerlad,

& Stal, 1996) – pattern-based classes are more change-prone. In saying that however, we need to be mindful of the possibility that poor developer practice of easy fixes and irregular maintenance practice (the phenomenon we have just described) may be at the heart of this result.

5.3.3 PATTERN-BASED CLASSES AND FAULT-PRONENESS

The next hypothesis we explored was whether pattern-based classes were less fault-prone than other (non-pattern) classes when considering all faults to be equal (i.e., the relative size of the change needed to rectify the fault in terms of lines of code was not considered). For the subset of 658 classes representing pattern-based classes, the average number of faults per class was 0.31. The remaining classes in the system (i.e., non-pattern classes) including those classes to which no fault could be attributed had an average number of faults per class of 0.21. However, when we consider only non-pattern classes that actually contained at least one fault, the average number of faults per class and the median values were found to be similar.

Table 5-3 shows the mean and median number of faults for pattern-based classes and for non-pattern classes that contained at least one fault. Pattern-based classes are marginally more fault-prone than non-pattern classes (1.77 compared with 1.75), while the median values are the same (1.00).

Pattern-based classes with faults; mean number of faults	1.77
Pattern-based classes with faults; median value of faults	1.00
Non-pattern classes with faults; mean number of faults	1.75
Non-pattern classes with faults; median value of faults	1.00

TABLE 5-3 SUMMARY VALUES FOR FAULTS IN CLASSES THAT CONTAIN FAULTS

One revealing feature of the data not illustrated by Table 5-3 is the relative likelihood of a class containing at least one fault and the distribution of faults in either type of class.

Scrutiny of the data revealed that while non-pattern classes tended to comprise the most fault-prone classes, i.e., classes with a large number of faults, the set of pattern-based classes were far more likely to comprise a single fault across many classes. In

other words, for the set of non-pattern, faulty classes, a high proportion of the faults were concentrated in a small number of classes; on the other hand, there was a more even distribution of faults in the pattern-based classes.

One positive and tentative message to emerge from this analysis is that while implementing patterns does not preclude the possibility of faults, there is some evidence to suggest that it limits the frequency of faults in pattern-based classes when compared with non-pattern classes.

5.3.4 PATTERN ANALYSIS

A research question that arises from this study is whether certain types of the thirteen patterns studied had a higher propensity for faults than others.

Table 5-4 shows the breakdown of faults by design pattern and frequency of design patterns. A small number of design patterns had a significantly higher fault-proneness than others, specifically the Adaptor, Method and Singleton patterns. Only four classes were found to be part of the Creator pattern. The simplistic nature of this pattern (comprising only a few lines of code and whose function is to merely instantiate another object) is one suggestion for the relative lack of faults in the case of this pattern.

Pattern	Classes	No. Faults	Faults/class
Adaptor	74	60	0.81
Builder	64	13	0.20
Command	27	2	0.07
Creator	4	0	0.00
Factory	97	19	0.20
Method	8	7	0.88
Filter	16	2	0.13
Iterator	0	0	0.00
Proxy	0	0	0.00

Singleton	170	68	0.40
State	32	3	0.09
Strategy	101	20	0.20
Visitor	66	4	0.06

TABLE 5-4 FAULTS PER DESIGN PATTERN

One explanation for the Adaptor pattern to contain a relatively high number of faults might be due to the high coupling nature of the pattern by definition. It has to provide a common interface to many classes and this might be the source of the problem in this case. Equally, a Singleton pattern might well be considered a 'key' class in the system because of the role that a Singleton class normally fulfils (typically logging and help features etc); this might actually have the opposite effect to that intended. Singleton classes could be the subject of relatively high maintenance/change activity, one result of which is a higher propensity for faults. Of the patterns for which occurrences were found, only the Creator pattern was found to be without any incidence of faults (this might be expected due to the simplistic nature of the Creator pattern).

FIgure 5-15 shows the box plot for pattern-based and non-pattern classes for classes containing faults and shows the lower and upper quartiles and median to be higher for non-pattern classes. The upper extremes and highest outlier however, are also higher for the non-pattern classes. This supports the observation made in the data that while non-pattern classes tend to have the highest number of faults, concentrated in a small number of classes, faults in pattern-based classes are more evenly distributed.

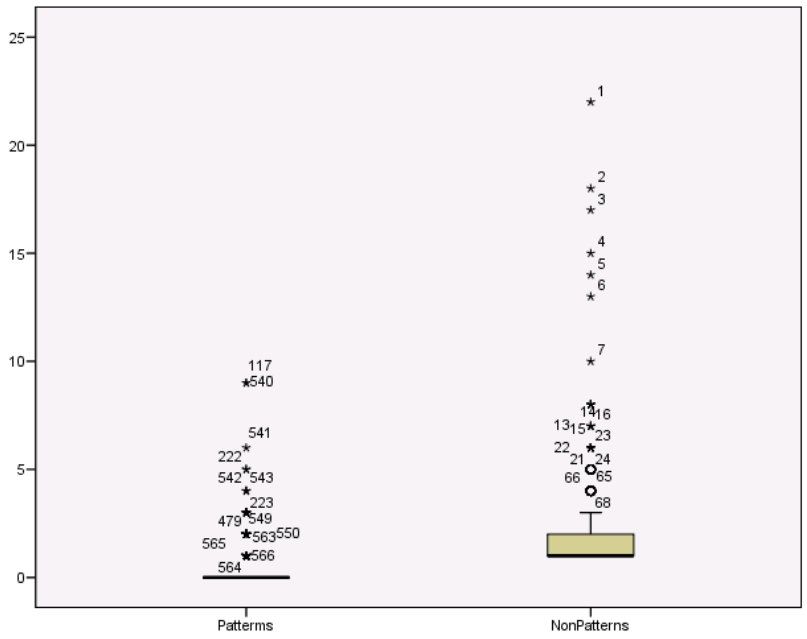

FIGURE 5-15 NUMBER OF FAULTS PER CLASS

As part of the analysis, we correlated number of faults in all classes (whether they had zero or more faults) with the number of design patterns a class was a participant of (in most cases, this would be zero). Using both Pearson's and Spearman's (parametric and non-parametric correlations, respectively) we found a significant correlation with Pearson's of 0.028, significant at the 5% level and a Spearman's Rank correlation of 0.047, significant at the 1% level. In other words, there is a strong likelihood that if a class is pattern-based, then it will contain at least one fault. We temper that claim with the remark that such a class is unlikely to contain more than a single fault. Inspection of the data revealed that the most fault-prone pattern-based classes contained 9 faults; two classes shared this number of faults; this compares with 22 faults for the most fault-prone non-pattern class.

5.3.5 FAULT-PRONENESS AND CHANGE SIZE

One question that complements the preceding analysis is whether the pattern classes are more fault-prone when, instead of considering each fault as equal, we consider a fault by the number of lines of code changed (either added, deleted or modified) in order to rectify the fault. Additionally, whether the effort required to fix a fault differs

between pattern-based classes and non-pattern classes for faulty classes in each category. We thus explore whether classes (pattern-based or non-pattern) were less fault-prone, when we consider the size of the fault repair, itself expressed in terms of LOC added, deleted or modified.

5.3.6 CHANGE ANALYSIS

Inspection of the average number of lines added, modified and deleted for faults in the entire system (pattern-based and non-pattern) found that, on average, a fault fix required 7.6 added LOC, 0.8 of a modified LOC and 0.8 deleted LOC.

Interestingly, the most common corrective action in order to correct a fault was to add more code around existing code rather than amending or deleting existing code. In fact, it appears relatively uncommon for large amounts of existing code to be changed when correcting faults. Considering pattern-based classes specifically, we found that on average, such a class was more likely to have LOC added, modified and deleted in order to rectify a fault when compared with a non-pattern class. Table 5-5 shows the breakdown of lines added, deleted and modified for faults for pattern-based classes and non-pattern classes.

Change/Class type	**Pattern-based**	**Non-pattern**
Ave. lines add.	8.30	7.55
Ave. lines mod.	1.57	0.73
Ave. lines del.	0.83	0.76
Ave. lines (all.)	10.70	9.04

TABLE 5-5 AVERAGE LINES ADDED, MODIFIED AND DELETED PER FAULT

Table 5-6 shows the breakdown of added, modified and deleted LOC for the two categories (pattern-based and non-pattern classes). When we consider only the set of pattern-based and non-pattern classes that contained faults, the average LOC affected per class was generally higher for non-pattern classes (except when considering the

number of lines modified, in which case they were similar). This suggests that despite a pattern-based class being more likely to contain a fault, the resulting corrective action was generally less effort in terms of total LOC changed (added, modified and deleted) for that pattern-based class.

The median values of added and modified LOC, however, were all higher for pattern-based classes. The median value of added lines, in particular, shows that pattern-based classes tended to have a larger numbers of LOC added more frequently and consistently in order to correct a fault (50% of all fault fixes in this category required 39 or more lines of code added), despite overall having less LOC added. This result suggests, as per the distribution of faults, that the large changes are focused on a small number of non-pattern classes.

One positive outcome of the analysis might be that from a consistency perspective, since pattern-based classes induce a more even distribution of faults, the effort to fix that fault is correspondingly distributed. This claim needs to be tempered with the view that it is better for a large number of faults to be concentrated in a small area where they can be tackled, rather than uniformly distributed over a larger number of classes.

Category/Change type	Add.	Mod.	Del.	Total
Pattern-based classes with faults; average number of lines	55.27	27.14	7.51	61.16
Pattern-based classes with faults; median value of lines	39.00	11.00	3.00	40.00
Non-pattern classes with faults; average number of lines	73.15	26.51	11.74	75.76
Non-pattern classes with faults; median value of lines	35.50	6.00	3.00	31.00

TABLE 5-6 AVERAGE AND MEDIAN LOC AFFECTED (FAULTY CLASSES)

Figure 5-16 shows the box plot for pattern-based classes and non-pattern classes that contained changed LOC (i.e., the total number of lines of code either added, modified or deleted) to rectify faults. It shows that the lower quartile, upper quartile and median were all higher for non-pattern classes.

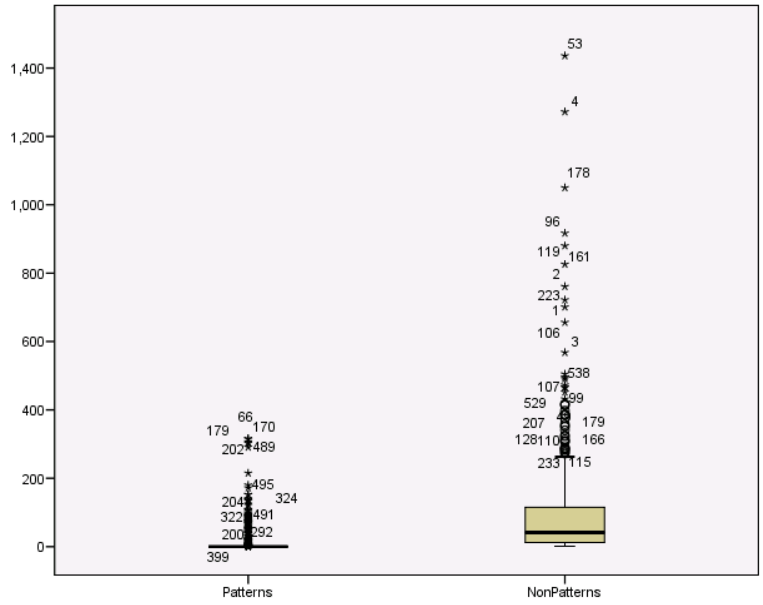

FIGURE 5-16 LINES OF CODE CHANGED

The upper extremes and highest outlier are also higher for the non-pattern classes. So, while a pattern class is more likely to require a line of code to be changed (either through its addition, change or removal) to fix a fault, classes with the highest number of LOC changed (to fix faults) were to non-pattern classes.

To further our analysis, we correlated the total number of lines of code that were 'affected' by a fault (i.e., sum of added, modified or deleted LOC) with the number of design patterns a class was a participant of – to determine whether design pattern participants required more effort (in terms of lines of code) to rectify faults than non-participants. This included all classes, including those that had no faults corrected in the studied period. Pearson's correlation value was found to be insignificant at 0.005 but Spearman's rank correlation was found to be 0.05, significant at the 1% level. In other words, pattern-based classes generally tended to affect more lines of code than non-pattern classes; however, the average values from Table 5-5 reflect the small number of non-pattern classes accounting for a large total number of affected lines centered around that small set of classes. The largest number of affected lines for pattern-based classes was 316, compared with 1436 for the corresponding non-pattern class.

5.3.7 INDIVIDUAL PATTERNS

Inspection of the average number of faults per class for each design pattern shows a small number of design patterns with a significantly higher fault-proneness than others, especially the Adaptor, Method and Singleton patterns.

Table 5-7 shows the breakdown of lines affected per fault, by design pattern. There is a clear trend for added LOC to be the key change type for the majority of patterns. It would seem that one effect of a fault occurring was not the modification of existing LOC, but the addition of further LOC; this is more in keeping with the addition of functionality than correction of existing.

Design Pattern	**Classes**	**Ave. add.**	**Ave. mod.**	**Ave. del.**
Adaptor	**74**	**16.66**	**4.69**	**2.34**
Builder	64	5.22	0.03	0.16
Command	27	0.00	0.19	0.04
Creator	4	0.00	0.00	0.00
Factory	97	8.77	0.06	0.18
Method	**8**	**21.50**	**0.00**	**0.50**
Filter	16	0.00	0.00	0.13
Iterator	0	0.00	0.00	0.00
Proxy	0	0.00	0.00	0.00
Singleton	**170**	**12.72**	**2.61**	**1.08**
State	32	0.53	0.81	0.06
Strategy	101	3.57	1.44	0.87
Visitor	66	1.89	0.03	0.45

TABLE 5-7 LINES AFFECTED BY FAULT

5.3.8 ELIMINATING THE EFFECT OF CHANGE

From previous studies, we know that classes that are more change-prone become more fault-prone; so in the context of this study one important question remains: are the design pattern participant classes more fault-prone simply because they are more change-prone or is there something inherent in design patterns and their usage that make them more fault-prone (over and above what we would expect when considering their increased level of change activity)?

To address this question we first confirm that, in line with other studies, there is a correlation between change propensity and fault-proneness in the studied system. We then use the Spearman and Pearson correlations to verify that there is a correlation between the number of changes made to a class (excluding changes made to rectify a fault) and the number of changes made to a class to rectify a fault. Pearson's correlation gave a coefficient of 0.611 and the Spearman correlation a coefficient of 0.480, both significant at the 0.01 level. This confirms that, as with many other studies, there is a correlation between change propensity and fault propensity in this system. As we conclude that change-proneness is a contributory factor to fault-proneness, we now attempt to eliminate the effect of change on our results to see whether design pattern classes have greater fault-proneness than we would expect for classes with this level of propensity for change.

To do this, we calculated the average number of faults per change for non-design pattern classes and compared it to the average number of faults per change for design pattern classes. For non-design pattern classes, the average number of faults per change was 0.16. For design pattern classes, the average number of faults per change was 0.14. This shows that non-design pattern classes are marginally more fault-prone than design pattern classes when eliminating the effect of change (which we know to be a contributing factor to fault propensity).

We can therefore conclude that while design pattern classes are more fault-prone than classes that are not participants in design patterns, this is largely due to the change propensity of design patterns classes, rather than any other contributory factor inherent in design patterns and their usage.

5.4 DISCUSSION

One question raised by the preceding study is why pattern-based classes should be the subject of such fault characteristics? After all, the whole purpose of patterns is to promote reuse and a common understanding of the design (Gamma, 1995).

Paradoxically, the very strength of patterns seems to be its undoing. To explain. When manually inspecting a sample of the pattern classes that had been subject to higher than normal rate of faults, the authors observed that many of these classes contained implementation code that should not have been the responsibility of the pattern class to provide. In other words, we suspect that because of the strong familiarity with pattern classes, system developers naturally choose these pattern classes as the focal point for making changes, instead of immediately surrounding classes (which would have been inline with the intention of the original design). This then leads to a higher rate of faults within those classes. This was particularly the case for the highest fault-prone patterns: Adaptor, Method and Singleton. Ironically, these types of pattern are characterised by coupling (in the case of the Adaptor pattern), inheritance in the case of the Method pattern) and potential large class size (in the case of the Singleton pattern). All three facets of object-oriented have been associated with a higher propensity for faults. Our study therefore may shed some light on why pattern-based classes have a higher propensity for faults – it is simply due to familiarity with those classes to developers which make them easy to adapt and change. Based on this premise, developers should take particular care with the design of class structures surrounding existing design patterns and ensure that new (added) code does not bloat the code in classes related to certain patterns. Equally, that coupling and class size is monitored very closely. As interestingly, pattern-based classes seem equally susceptible, as any type of class, to the causal relationship between changes and faults.

For a study of this type, the threats to its validity need to be considered. First, we have to consider whether the results from a study of one system can be generalised; future replications of this study over different systems will add to the validity of this study, but we feel that as the study itself is a replication (and extension) of previous studies by Bieman et al., (Bieman, Jain, & Yang, 2001) on a different system.

Second, the study only considers a subset of the commonly used design patterns; however, the design patterns selected are in accordance with a previous study by Bieman et al., and we feel offer a broad cross-section of the types of design pattern.

Third, the manual identification of design patterns may result in some undetected design pattern participants. However, it is likely that automation of pattern identification would also require significant human involvement and effort and, ultimately, this would be as likely to cause identification errors.

Finally, one criticism of the use of any fault is that it does not represent the full set of faults that had occurred during the period studied; often faults remain unreported for a multitude of reasons. This could have a significant influence on the conclusions we have drawn. However, a strong team ethos and set of standards were applied during the period studied; an automated, enforced check-in policy was implemented in which any change to the source code had to be accompanied by a description of the reason for that change (whether enhancement, fault or other change activity); we therefore have no reason to believe that faults fell into the unreported category.

5.5 CONCLUSIONS

In this chapter, we have described a study of change and fault-proneness in commercial software with respect to design pattern usage. The aims of this study were:

1. To understand how design patterns were applied in commercial software.
2. To determine which patterns were most common and how often are they applied.
3. To see whether design pattern participation resulted in less change and fault-prone classes.

We studied WebCSC for a 24 month period and identified design pattern participants by manually inspecting the source code and extracted change and fault data from the source control system in order to determine whether pattern-based classes were more fault-prone than non-pattern classes. Faults were measured by considering the number of changes made to a class to rectify a fault and also the size of the change (in terms of number of lines of code added, deleted or modified in a class).

To summarise our findings, in the context of the system studied, we have shown that:

1. Pattern-based classes were more change-prone than non-pattern classes, in terms of number of changes made to a class
2. Pattern-based classes were more fault-prone than non-pattern classes, in terms of the number of changes made to a class
3. Pattern-based classes were also more fault-prone than non-pattern classes, in terms of the size of the changes required to fix a fault
4. Adaptor, Method and Singleton patterns were found to be the most fault-prone patterns, both in terms of the number of changes made, and the size of the changes required to fix a fault
5. The primary reason for pattern-based classes being more fault-prone lay with the propensity for change in design pattern classes
6. We also revealed an interesting trend that the most common corrective action, when dealing with a fault, was the addition of new lines of code around existing code, rather than the modification or removal of existing code

6 CHAPTER 6 – CONCLUSIONS AND FUTURE WORK

6.1 INTRODUCTION

In this chapter we conclude with a statement of the achievements of this Thesis. First, in Section 6.2, we repeat the objectives of the research, originally set out in Chapter 1, before describing how the research documented in this thesis supports the objectives. Second, in Section 6.3, we list the contributions that this thesis adds to the software engineering field. Third, in Section 6.4, the achievements are discussed at a personal level. Finally, in Section 6.5, we discuss possible future work.

6.2 OBJECTIVES

The objectives of this Thesis, as stated in Chapter 1, are:

1. To understand how refactorings are applied in commercial software. Are refactorings regularly applied, and if so what types of refactorings are most common? Are refactorings inter-dependent (that is, if refactoring *x* is applied is there a likelihood that refactoring *y* will also be applied?). Does the application of refactorings in commercial software differ from what we have previously seen in open source studies?

2. Second, to understand how design patterns are applied in commercial software. Which patterns are most common? How often are they applied? Does design pattern participation result in less change-prone and fault-prone classes?

3. Are the key aspects of the object-oriented design context: inheritance, coupling and cohesion, and size, contributory factors of fault and change propensity? Are there certain properties or characteristics of the design context that cause a higher propensity for fault or change-proneness more than others?

In Chapter 3 we empirically investigated the application of refactorings in WebCSC. We documented the frequency of which 15 key refactorings were applied and showed the most common and proposed why the least common were so. We showed inter-dependencies between some types of refactorings and we showed a common trend between the application of refactorings in WebCSC and several other refactoring studies that used open-source software. We showed the impact test code had during the

refactoring process, showing both the number and type of refactorings applied to test code. We feel that the conclusions drawn from Chapter 3 support objective (1).

In Chapter 5 we empirically investigated the application of design patterns in WebCSC, documenting the frequency and popularity of 13 key design patterns. We went further to show design pattern participants tended to have a higher propensity for change than non-participants. We showed that design pattern participants tended to have a higher propensity for faults than non-participants, and the primary reason for this was the effect of greater change-proneness. We feel that the conclusions drawn from Chapter 5 support objective (2).

In Chapter 4 we empirically investigated the effect the design context of a class had on change and fault propensity. Specifically, we considered inheritance characteristics, coupling and cohesion properties, and size. We showed that inheritance characteristics played a role in the change and fault propensity of a class, in terms of the depth of inheritance that a class exists in, and the number of child classes a class possesses. We showed that coupling was a contributory factor to change and fault-proneness, afferent coupling being a stronger indicator of change and faults than efferent coupling. We also showed that size, when measured by lines of code, number of methods, and number of fields, was an influencing factor on change and fault propensity. We feel that the conclusions drawn from Chapter 4 support objective (3).

To conclude, we feel that across the four empirical investigations documented as part of this Thesis we have achieved the four objectives that we set out to meet. We have identified a number of aspects, ranging from specific details of the design context, to a common maintenance process, that are contributory factors to the propensity for change and faults in the studied system.

6.3 CONTRIBUTIONS

The contribution of this thesis spans several research strands. First, we have added information about how refactorings are applied to commercial software, and compared the refactoring trends in commercial software to open-source refactoring studies. Second, we have detailed how design patterns are applied to commercial software and shown that design patterns participants are more prone to change and faults than non-

participants. Finally, we have described detailed information about the design context of commercial software: inheritance characteristics, coupling and cohesion properties and size, and we have shown that certain aspects of the design context are contributory factors to change and fault-proneness, such as a deep or wide inheritance hierarchy, high coupling, and a large size. We have identified a number of contributory factors to change and fault-proneness in a large-scale, commercial, object-oriented software system. The study presented replicated earlier studies in certain instances and extended those studies through the exploration of faults.

6.4 PERSONAL ACHIEVEMENTS

The most important achievement, in undertaking a PhD, is to learn how to research in the correct manner, and now, nearing the end of the process, I feel a different person academically and professionally as a consequence. To be able study a small and niche area of a broad subject to a fine and meticulous detail requires a degree of patience, self-motivation and discipline seen in few other endeavours. The colossal number of hours spent collecting and analyzing data, searching for a lead or an idea, combined with the number of false leads and research dead-ends can be painstaking. However, one lesson that is quickly learned is that when faced with a result that was not expected (or wanted) this can often lead to a more interesting study. Research can be an isolating and lonely experience, but this is balanced with the rewarding experience of sharing and debating the research in front of an interested, knowledgeable and enthusiastic audience at international research conferences, and speaking at three of these conferences has undoubtedly been one of the highlights of my experience. Above all, completing this Thesis has been epic, and it has left me with an enormous sense of achievement and satisfaction. Finally, as a consequence of the research undertaken I am left with a more intimate knowledge of the WebCSC system than, I suspect, anyone past, present or future; perhaps an even a deeper knowledge than many can even imagine.

6.5 FUTURE WORK

In terms of future work, the design patterns study could be extended by detecting other common design patterns and replicating the study using additional proprietary or open-source software. For refactoring, increasing the number or refactoring types detected

by the tool and increasing the number of versions considered could extend this study. Replicating the study over other systems would also add further validity to the results. Another research question that remains open is whether refactorings are embedded within other refactorings and identifying these embedded refactorings into this study may add an interesting perspective. In our research we have touched on the impact of test classes, specifically related to refactoring and fault and change-proneness, but this is a far broader topic; investigating the different forms of testing: state-based compared with interaction-based and their relevant cost and benefits, for example, would be an interesting study.

Other testing areas include a detailed analysis of the co-evolution of production and test code from a change and fault perspective. Other areas of a classes' design context could be considered, in relation to change and fault-proneness, such as polymorphism, encapsulation; the use of different metrics to measure, for example, cohesion would add an interesting comparison to the results in this thesis. Other common object-oriented design techniques, such as dependency injection, could also be studied as potential contributory factors to change and fault-proneness.

Finally, the system used in this study is an ongoing and ever-developing artefact and from an evolutionary perspective it would be interesting to observe whether the same trends recur as the system ages further.

APPENDIX A – LIST OF PUBLICATIONS

This section lists the publications that form the basis of the research in this thesis.

2009

Gatrell, M., & Counsell, S. (2009). Empirical Support for Two Refactoring Studies Using Commercial C# Software. *13th International Conference on Evaluation & Assessment in Software Engineering.* Durham.

Gatrell, M., Counsell, S., & Hall, T. (2009). Design patterns and change-proneness: a replication using proprietary C# software. *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'09).* Lille.

2010

Gatrell, M., & Counsell, S. (2010). Size, Inheritance, Change and Fault-proneness in C# software. *The Journal of Object Technology , 9* (5), pp29-54.

2011

Gatrell, M., & Counsell, S. (2011). Design Patterns and Fault-Proneness: A Study of Commercial C# Software. *Fifth IEEE International Conference on Research Challenges in Information Science.* Guadeloupe.

2012

Gatrell, M., & Counsell, S. (2012). Faults and their Relationship to Implemented Patterns, Coupling and Cohesion in Commercial C# Software. *(To appear in) International Journal of Information System Modeling and Design*.

APPENDIX B – SOURCE CODE FOR REFACTORING TOOL

Report UI – class: RefactoringMinor

The RefactoringMinor class is the main UI and allows certain parameters to be set, controls the execution, and displays the results.

```
public partial class RefactoringMinor : Form
{

    DataSet _ds = null;

    public RefactoringMinor()
    {
        InitializeComponent();

    }

    /// <summary>
    /// Main execution
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void btnGo_Click(object sender, EventArgs e)
    {
        // Reset model and UI
        Cursor.Current = Cursors.WaitCursor;
        lblOverallProgress.Text = "Preparing for new analysis...";
        Application.DoEvents();
        IModel model = new Model();
        model.ClearModel();
        int version = 0;
        int numberOfErrors = 0;
        prgOverallProgress.Value = 0;
        prgOverallProgress.Maximum = versionFolders.Length;

        // Get source code versions folders
        string[] versionFolders = Directory.GetDirectories(txtSourceLocation.Text);
        IResults results = new Results();

        // Loop through each version
        foreach (string versionFolder in versionFolders)
        {
            version++;

            lblOverallProgress.Text = "Analysing version " + version.ToString() + "...";
            Application.DoEvents();

            // Get all source code files from a specific version
            string[] files = Directory.GetFiles(versionFolder, "*.cs",
                  SearchOption.AllDirectories);
            Parser parser = new Parser(versionFolder);

            prgVersionProgress.Value = 0;
            prgVersionProgress.Maximum = files.Length;

            //For each source code file in a version
            foreach (string file in files)
            {
                try
                {
                    lblVersionProgress.Text = "Analysing " + file.ToString() + "...";
                    Application.DoEvents();

                    // parse the source code file and populate the model
                    parser = new Parser(file);
                    CompilationUnitNode code = parser.Parse(file);
                    model.Populate(code, version);

                    prgVersionProgress.Value++;
```

```
                }
                catch { numberOfErrors++; }
            }
            // If it is at least second version analysed, compare
            // with the preceding version to detect refactorings
            if (version > 1)
            {
                try
                {
                    MineForRefactorings(results, version);

                    // If we are complete, display results
                    if (version == files.Length)
                    {
                        _ds = results.GetResults(version);
                        dataGrid.DataSource = _ds.Tables[0];
                    }
                    // otherwise delete previous version and carry on
                    // this is to reduce the database size as it gets too
                    // big to query otherwise
                    else
                    {
                        model.DeleteVersion(version - 2);
                        DataSet ds = results.GetResults(version);
                        DataRow dr = ds.Tables[0].Rows[0];
                        _ds.Tables[0].ImportRow(dr);
                    }
                    dataGrid.Refresh();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
            }

            prgOverallProgress.Value++;
        }
        lblOverallProgress.Text = "Idle...";
        lblVersionProgress.Text = "Idle...";

        Cursor.Current = Cursors.Default;

        MessageBox.Show("Completed.  Number of errors: "
           + numberOfErrors.ToString(),
           "Completed",
           MessageBoxButtons.OK,
           MessageBoxIcon.Information);
    }

    /// <summary>
    /// Call each refactoring detection class
    /// </summary>
    /// <param name="results"></param>
    /// <param name="version"></param>
    private void MineForRefactorings(IResults results, int version)
    {
        // Add Parameter
        DetectRefactorings(new AddParameter(), results, version);
        // Remove Parameter
        DetectRefactorings(new RemoveParameter(), results, version);
        // Hide Method
        DetectRefactorings(new HideMethod(), results, version);
        // Rename Field
        DetectRefactorings(new RenameField(), results, version);
        // Rename Method
        DetectRefactorings(new RenameMethod(), results, version);
        // Move Field
        DetectRefactorings(new MoveField(), results, version);
        // Move Method
        DetectRefactorings(new MoveMethod(), results, version);
        // Pull Up Field
        DetectRefactorings(new PullUpField(), results, version);
        // Pull Up Method
        DetectRefactorings(new PullUpMethod(), results, version);
        // Extract Super Class
        DetectRefactorings(new ExtractSuperClass(), results, version);
```

```
        // Encapsulate Fields
        DetectRefactorings(new EncapsulateField(), results, version);
        // Encapsulate Downcast
        DetectRefactorings(new EncapsulateDownCast(), results, version);
        // Extract Subclass
        DetectRefactorings(new ExtractSubClass(), results, version);
        // Pull Down Field
        DetectRefactorings(new PullDownField(), results, version);
        // Pull Down Method
        DetectRefactorings(new PullDownMethod(), results, version);
    }

    /// <summary>
    /// Detect refactorings, record any results in the database
    /// </summary>
    /// <param name="refactoringRule"></param>
    /// <param name="results"></param>
    /// <param name="version"></param>
    private void DetectRefactorings(IRefactoringRule refactoringRule, IResults results, int
version)
    {
        lblVersionProgress.Text = "Detecting " +
          refactoringRule.GetType().Name  +
          " refactorings...";
        Application.DoEvents();

        List<IRefactoringResult> refactoringResults =
            refactoringRule.DetectRefactorings(version - 1, version);

        foreach (IRefactoringResult refactoringResult in refactoringResults)
        {
            results.AddRefactoringResult(refactoringResult, version);
        }
    }

    /// <summary>
    /// Displays a folder dialog to locate source folders
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void btnFolderBrowser_Click(object sender, EventArgs e)
    {
        folderBrowserDialog1.ShowDialog();
        txtSourceLocation.Text = folderBrowserDialog1.SelectedPath;
    }

    /// <summary>
    /// Gets and displays full results from the database
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void btnGetResultsDetail_Click(object sender, EventArgs e)
    {
        IResults results = new Results();

        cboVersions.DataSource = results.GetVersions().Tables[0];
        cboVersions.DisplayMember = "Version";

        if (cboVersions.Text != string.Empty && cboVersions.Text != null)
        {
            dataGridDetail.DataSource =
               results.GetResultsDetail(int.Parse(cboVersions.Text)).Tables[0];
        }
    }

}
```

Model Interface

```
public interface IModel
{
    /// <summary>
    /// Populates the model
    /// </summary>
    /// <param name="codeDOM"></param>
    /// <param name="version"></param>
    void Populate(CompilationUnitNode codeDOM, int version);

    /// <summary>
    /// Stores the number of classes in a version
    /// </summary>
    /// <param name="version">The Version number</param>
    /// <returns>The number of classes in a version</returns>
    int NumberOfClasses(int version);

    /// <summary>
    /// Returns the number of methods in a version
    /// </summary>
    /// <param name="version">The version number</param>
    /// <returns>The number of methods in a specified version</returns>
    int NumberOfMethods(int version);


    /// <summary>
    /// Deletes a version
    /// </summary>
    /// <param name="version">The version number</param>
    void DeleteVersion(int version);

    /// <summary>
    /// Clears the model
    /// </summary>
    void ClearModel();
}
```

Model – Concrete class

The Model is responsible for populating the database model representing the parsed source code

```
public class Model : IModel
{

    #region IPopulateModel Members

    /// <summary>
    /// Populates the model in the database
    /// </summary>
    /// <param name="codeDOM">In memory representation of the code</param>
    /// <param name="version">Version number</param>
    void IModel.Populate(CompilationUnitNode codeDOM, int version)
    {
        // Create database connection/transaction
        Database        database    = DatabaseFactory.CreateDatabase();
        DbConnection    connection  = database.CreateConnection();
        connection.Open();
        DbTransaction   transaction = connection.BeginTransaction();

        try
        {
            // loop through namespaces
            foreach (NamespaceNode namespaceNode in codeDOM.Namespaces)
            {
                // loop through classes in each namespace
                foreach (ClassNode classNode in namespaceNode.Classes)
                {
                    // get the base class
                    string baseClass = string.Empty;
                    if (classNode.BaseClasses.Count > 0)
                    {
                        baseClass =
                          ((TypeNode)classNode.BaseClasses[0]).Identifier.QualifiedIdentifier;
                    }

                    // add the class
                    int classId = AddClass(database,
                                           connection,
                                           transaction,
                                           namespaceNode.Name.QualifiedIdentifier + "." +
                                               classNode.FriendlyName,
                                           version,
                                           baseClass);

                    // add the fields for that class
                    foreach (FieldNode fieldNode in classNode.Fields)
                    {
                        AddField(database, connection, transaction,
                                fieldNode.FriendlyName,
                                ((TypeNode)fieldNode.Type).Identifier.QualifiedIdentifier,
                                classId,
                                fieldNode.Modifiers.ToString());
                    }

                    // add the methods for that class
                    foreach (MethodNode methodNode in classNode.Methods)
                    {
                        int methodID = AddMethod(database,
                                                 connection,
                                                 transaction,
                                                 methodNode.FriendlyName,
                                                 methodNode.Signature,
                                                 classId,

((TypeNode)methodNode.Type).Identifier.QualifiedIdentifier,
                                                 methodNode.Modifiers.ToString(),
                                                 methodNode.ToSource());
                        // add the parameters for that method
                        foreach (ParamDeclNode parameter in methodNode.Params)
                        {
```

```
                            AddParameter(database,
                                         connection,
                                         transaction,
                                         parameter.Name,

((TypeNode)parameter.Type).Identifier.QualifiedIdentifier,
                                         methodID);
                        }
                    }
                }
            }
            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }

    #endregion

    #region Data accessors

    /// <summary>
    /// Adds the class.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="transaction">The transaction.</param>
    /// <param name="className">Name of the class.</param>
    /// <param name="version">The version.</param>
    /// <param name="superClass">The super class.</param>
    /// <returns></returns>
    public int AddClass(Database database, DbConnection connection ,
                        DbTransaction transaction, string className,
                        int version, string superClass)
    {
        if (database    == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection  == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand       command     = connection.CreateCommand();
        command.CommandType         = CommandType.StoredProcedure;
        command.CommandTimeout = 0;

        DbParameter    nameParam    = command.CreateParameter();
        nameParam.ParameterName     = "@Name";
        nameParam.DbType            = DbType.String;
        nameParam.Value             = className;
        command.Parameters.Add(nameParam);

        DbParameter     versionParam    = command.CreateParameter();
        versionParam.ParameterName      = "@Version";
        versionParam.DbType             = DbType.Int16;
        versionParam.Value              = version;
        command.Parameters.Add(versionParam);

        DbParameter superClassParam     = command.CreateParameter();
        superClassParam.ParameterName   = "@SuperClass";
        superClassParam.DbType          = DbType.String;
        superClassParam.Value           = superClass;
        command.Parameters.Add(superClassParam);

        command.CommandText = "spAddClass";

        object retValue = database.ExecuteScalar(command, transaction);

        return Convert.ToInt32(retValue);
```

```
    }

    /// <summary>
    /// Adds the field.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <param name="fieldName">Name of the field.</param>
    /// <param name="fieldType">Type of the field.</param>
    /// <param name="classID">The class ID.</param>
    /// <returns></returns>
    public int AddField(Database database, DbConnection connection ,
          DbTransaction transaction, string fieldName,
          string fieldType, int classID, string modifier)
    {
        if (database    == null)
    throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection  == null)
    throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
    throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand       command     = connection.CreateCommand();
        command.CommandType         = CommandType.StoredProcedure;
        command.CommandTimeout = 0;

        DbParameter    nameParam    = command.CreateParameter();
        nameParam.ParameterName     = "@Name";
        nameParam.DbType            = DbType.String;
        nameParam.Value             = fieldName;
        command.Parameters.Add(nameParam);

        DbParameter     versionParam    = command.CreateParameter();
        versionParam.ParameterName      = "@Type";
        versionParam.DbType             = DbType.String;
        versionParam.Value              = fieldType;
        command.Parameters.Add(versionParam);

        DbParameter superClassParam     = command.CreateParameter();
        superClassParam.ParameterName   = "@ClassID";
        superClassParam.DbType          = DbType.Int32;
        superClassParam.Value           = classID;
        command.Parameters.Add(superClassParam);

        DbParameter modifierParam = command.CreateParameter();
        modifierParam.ParameterName = "@Modifier";
        modifierParam.DbType = DbType.String;
        modifierParam.Value = modifier;
        command.Parameters.Add(modifierParam);

        command.CommandText = "spAddField";

        object retValue = database.ExecuteScalar(command, transaction);

        return Convert.ToInt32(retValue);
    }

    /// <summary>
    /// Adds the method.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <param name="methodName">Name of the method.</param>
    /// <param name="signature">The signature.</param>
    /// <param name="classID">The class ID.</param>
    /// <returns></returns>
    public int AddMethod(Database database, DbConnection connection , DbTransaction
transaction, string methodName, string signature, int classID, string returnType, string
modifier, string code)
    {
        if (database    == null)
     throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection  == null)
     throw new ArgumentNullException("Connection was null when calling AddClass");
```

```
        if (transaction == null)
     throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand       command     = connection.CreateCommand();
        command.CommandType         = CommandType.StoredProcedure;
        command.CommandTimeout = 0;

        DbParameter    nameParam    = command.CreateParameter();
        nameParam.ParameterName     = "@Name";
        nameParam.DbType            = DbType.String;
        nameParam.Value             = methodName;
        command.Parameters.Add(nameParam);

        DbParameter     versionParam    = command.CreateParameter();
        versionParam.ParameterName      = "@Signature";
        versionParam.DbType             = DbType.String;
        versionParam.Value              = signature;
        command.Parameters.Add(versionParam);

        DbParameter superClassParam     = command.CreateParameter();
        superClassParam.ParameterName   = "@ClassID";
        superClassParam.DbType          = DbType.Int32;
        superClassParam.Value           = classID;
        command.Parameters.Add(superClassParam);

        DbParameter returnTypeParam = command.CreateParameter();
        returnTypeParam.ParameterName = "@ReturnType";
        returnTypeParam.DbType = DbType.String;
        returnTypeParam.Value = returnType;
        command.Parameters.Add(returnTypeParam);

        DbParameter modifierParam = command.CreateParameter();
        modifierParam.ParameterName = "@Modifier";
        modifierParam.DbType = DbType.String;
        modifierParam.Value = modifier;
        command.Parameters.Add(modifierParam);

        DbParameter modifierParam = command.CreateParameter();
        modifierParam.ParameterName = "@Code";
        modifierParam.DbType = DbType.Text;
        modifierParam.Value = code;
        command.Parameters.Add(modifierParam);

        command.CommandText = "spAddMethod";

        object retValue = database.ExecuteScalar(command, transaction);

        return Convert.ToInt32(retValue);
    }

    /// <summary>
    /// Adds a parameter
    /// </summary>
    /// <param name="database">The database</param>
    /// <param name="connection">Database connection</param>
    /// <param name="transaction">Database transaction</param>
    /// <param name="parameterName">Parameter Name</param>
    /// <param name="type">Parameter Type</param>
    /// <param name="methodID">Method ID</param>
    /// <returns></returns>
    public int AddParameter(Database database, DbConnection connection, DbTransaction
transaction, string parameterName, string type, int methodID)
    {
        if (database == null)
            throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
            throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
            throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;
        command.CommandTimeout = 0;

        DbParameter nameParam = command.CreateParameter();
        nameParam.ParameterName = "@Name";
```

```
        nameParam.DbType = DbType.String;
        nameParam.Value = parameterName;
        command.Parameters.Add(nameParam);

        DbParameter versionParam = command.CreateParameter();
        versionParam.ParameterName = "@Type";
        versionParam.DbType = DbType.String;
        versionParam.Value = type;
        command.Parameters.Add(versionParam);

        DbParameter superClassParam = command.CreateParameter();
        superClassParam.ParameterName = "@MethodID";
        superClassParam.DbType = DbType.Int32;
        superClassParam.Value = methodID;
        command.Parameters.Add(superClassParam);

        command.CommandText = "spAddParameter";

        object retValue = database.ExecuteScalar(command, transaction);

        return Convert.ToInt32(retValue);
    }

    #endregion

    #region IModel Members


    /// <summary>
    /// Returns the number of classes in a particular version.
    /// </summary>
    /// <param name="version">The version.</param>
    /// <returns></returns>
    int IModel.NumberOfClasses(int version)
    {
        Database        database    = DatabaseFactory.CreateDatabase();
        DbConnection    connection  = database.CreateConnection();
        connection.Open();
        DbTransaction   transaction = connection.BeginTransaction();

        try
        {
            DbCommand       command     = connection.CreateCommand();
            command.CommandType         = CommandType.StoredProcedure;
            command.CommandTimeout = 0;

            DbParameter versionParam     = command.CreateParameter();
            versionParam.ParameterName   = "@Version";
            versionParam.DbType          = DbType.Int32;
            versionParam.Value           = version;
            command.Parameters.Add(versionParam);

            command.CommandText = "spCountClasses";

            object retValue = database.ExecuteScalar(command, transaction);
            transaction.Commit();

            return Convert.ToInt32(retValue);
            }
        catch
        {
            transaction.Rollback();
            throw;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }

    /// <summary>
    /// Returns the number of methods in a particular version.
    /// </summary>
    /// <param name="version">The version.</param>
    /// <returns></returns>
```

```
    int IModel.NumberOfMethods(int version)
    {
        Database        database    = DatabaseFactory.CreateDatabase();
        DbConnection    connection  = database.CreateConnection();
        connection.Open();
        DbTransaction   transaction = connection.BeginTransaction();

        try
        {
            DbCommand       command     = connection.CreateCommand();
            command.CommandType         = CommandType.StoredProcedure;
            command.CommandTimeout = 0;

            DbParameter versionParam     = command.CreateParameter();
            versionParam.ParameterName   = "@Version";
            versionParam.DbType          = DbType.Int32;
            versionParam.Value           = version;
            command.Parameters.Add(versionParam);

            command.CommandText = "spCountMethods";

            object retValue = database.ExecuteScalar(command, transaction);
            transaction.Commit();

            return Convert.ToInt32(retValue);
            }
        catch
        {
            transaction.Rollback();
            throw;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }

    /// <summary>
    /// Clears the model.
    /// </summary>
    void IModel.ClearModel()
    {
        Database        database    = DatabaseFactory.CreateDatabase();
        DbConnection    connection  = database.CreateConnection();
        connection.Open();
        DbTransaction   transaction = connection.BeginTransaction();

        try
        {
            DbCommand       command     = connection.CreateCommand();
            command.CommandType         = CommandType.StoredProcedure;

            command.CommandText = "spClearModel";
            command.CommandTimeout = 0;

            object retValue = database.ExecuteNonQuery(command, transaction);
            transaction.Commit();
        }
        catch
        {
            transaction.Rollback();
            throw;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }


    /// <summary>
    /// Deletes a version
    /// </summary>
    /// <param name="version">The version</param>
```

```
    void IModel.DeleteVersion(int version)
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            DbCommand command = connection.CreateCommand();
            command.CommandType = CommandType.StoredProcedure;

            DbParameter versionParam = command.CreateParameter();
            versionParam.ParameterName = "@Version";
            versionParam.DbType = DbType.Int32;
            versionParam.Value = version;
            command.Parameters.Add(versionParam);

            command.CommandText = "spDeleteVersion";
            command.CommandTimeout = 0;

            object retValue = database.ExecuteNonQuery(command, transaction);
            transaction.Commit();
        }
        catch
        {
            transaction.Rollback();
            throw;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }

    #endregion
```

Base Refactoring Class

The [RefactoringName]Refactoring classes are responsible for executing the rule to detect instances of refactorings from the database model representing the parsed source co de, by calling the corresponding stored procedure

```
public abstract class BaseDatabaseModelRefactoring : IRefactoringRule
{
    /// <summary>
    /// Name of the stored procedure to call
    /// </summary>
    private string _storedProcedure = string.Empty;

    /// <summary>
    /// Accessor for the stored procedure name
    /// </summary>
    protected string StoredProcedure
    {
        get { return _storedProcedure; }
        set { _storedProcedure = value; }
    }

    /// <summary>
    /// Name of the refactoring
    /// </summary>
    private string _refactoringName = string.Empty;

    /// <summary>
    /// Accessor for the refactoring name
    /// </summary>
    protected string RefactoringName
    {
        get { return _refactoringName; }
        set { _refactoringName = value; }
    }

    protected abstract void SetStoredProcedureAndRefactoringName();

    #region IRefactoringRule Members

    /// <summary>
    /// Main method for detecting a refactoring.  Calls the stored procedure.
    /// </summary>
    /// <param name="version1">Base version number</param>
    /// <param name="version2">Next version to compare with</param>
    /// <returns></returns>
    List<IRefactoringResult> IRefactoringRule.DetectRefactorings(int version1, int version2)
    {
        this.SetStoredProcedureAndRefactoringName();

        if (_storedProcedure == string.Empty)
              throw new ArgumentException("The stored procedure has not been set for
                                           this refactoring rule.");
        if (_refactoringName == string.Empty)
              throw new ArgumentException("The refactoring name has not neen set for
                                           this refactoring rule.");

        Database        database    = DatabaseFactory.CreateDatabase();
        DbConnection    connection  = database.CreateConnection();
        DbCommand       command     = connection.CreateCommand();
        DbDataReader    reader      = null;

        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();
        List<IRefactoringResult> refactoringResults = new List<IRefactoringResult>();

        try
        {
            command.CommandType = CommandType.StoredProcedure;
            command.CommandTimeout = 0;
```

```
            command.CommandText = _storedProcedure;

            DbParameter version1Param = command.CreateParameter();
            version1Param.ParameterName = "@Version1";
            version1Param.DbType = DbType.Int32;
            version1Param.Value = version1;
            command.Parameters.Add(version1Param);

            DbParameter version2Param = command.CreateParameter();
            version2Param.ParameterName = "@Version2";
            version2Param.DbType = DbType.Int16;
            version2Param.Value = version2;
            command.Parameters.Add(version2Param);

            reader = (DbDataReader)database.ExecuteReader(command, transaction);

            while (reader.Read())
            {
                IRefactoringResult refactoringResult = new
                     RefacotoringResult(reader["classname"].ToString(), "", "",
                          _refactoringName, "A " + _refactoringName +
                          " refactoring was found in class: " +
                          reader["classname"].ToString().Trim() +
                          ", the subject of the refactoring was: " +
                          reader["subject"].ToString().Trim());
                refactoringResults.Add(refactoringResult);
            }

            reader.Close();
            reader.Dispose();
            command.Dispose();
            transaction.Commit();

            return refactoringResults;
        }
        catch
        {
            transaction.Rollback();
            throw;
        }
        finally
        {
            transaction.Dispose();
            connection.Dispose();
        }
    }

    #endregion
}
```

Refactoring Classes

The [RefactoringName]Refactoring classes are responsible for executing the rule to detect instances of refactorings from the database model representing the parsed source code, by calling the corresponding stored procedure.

```
/// <summary>
/// Detects whether the Add Parameter refactoring has been applied between two versions of
source code.
/// </summary>
public class AddParameter : BaseDatabaseModelRefactoring
{
    /// <summary>
    /// Sets the name of the stored procedure and refactoring.
    /// </summary>
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spAddParameterRefactoring";
        RefactoringName = "Add Parameter";
    }
}



/// <summary>
/// Detects whether an encapsulate downcast refactoring has been applied between two versions
of source code
/// </summary>
public class EncapsulateDownCast : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spEncapsulateDownCast";
        RefactoringName = "Encapsulate Downcast";
    }
}



/// <summary>
/// Detects whether an encapsulate field refactorings has been applied between two versions of
source code
/// </summary>
public class EncapsulateField : BaseDatabaseModelRefactoring
{
    /// <summary>
    /// Sets the name of the stored procedure and refactoring.
    /// </summary>
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spEncapsulateField";
        RefactoringName = "Encapsulate Field";
    }
}

/// <summary>
/// Detects whether an extract subclass refactoring has been applied between two versions of
source code
/// </summary>
public class ExtractSubClass : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spExtractSubClass";
        RefactoringName = "Extract SubClass";
    }
}
```

```
/// <summary>
/// Detects whether an extract super class refactoring has been applied between two versions
of source code
/// </summary>
public class ExtractSuperClass : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spExtractSuperClass";
        RefactoringName = "ExtractSuperClass";
    }
}


/// <summary>
/// Detects whether the Hide Method refactoring has been applied between two versions of
source code.
/// </summary>
public class HideMethod : BaseDatabaseModelRefactoring
{

    /// <summary>
    /// Sets the name of the stored procedure and refactoring.
    /// </summary>
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spHideMethod";
        RefactoringName = "Hide Method";
    }
}


/// <summary>
/// Detects whether a move field refactoring has been applied between two versions of source
code
/// </summary>
public class MoveField : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spMoveField";
        RefactoringName = "MoveField";
    }
}


/// <summary>
/// Detects whether a move field refactoring has been applied between two versions of source
code
/// </summary>
public class MoveMethod : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spMoveMethod";
        RefactoringName = "MoveMethod";
    }
}


/// <summary>
/// Detects whether a pull down field refactoring has been applied between two versions of
source code
/// </summary>
public class PullDownField : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spPullDownField";
        RefactoringName = "Pull Down Field";
    }
}


/// <summary>
/// Detects whether a push down method refactoring has been applied between two versions of
source code
/// </summary>
```

```
public class PullDownMethod : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spPullDownMethod";
        RefactoringName = "Pull Down Method";
    }
}

/// <summary>
/// Detects whether a pull up field refactoring has been applied between two versions of
source code
/// </summary>
public class PullUpField : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spPullUpField";
        RefactoringName = "PullUpField";
    }
}

/// <summary>
/// Detects whether a pull up method refactoring has been applied between two source code
versions
/// </summary>
public class PullUpMethod : BaseDatabaseModelRefactoring
{
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spPullUpMethod";
        RefactoringName = "PullUpMethod";
    }
}

/// <summary>
/// Detects whether the Remove Parameter refactoring has been applied between two versions of
source code.
/// </summary>
public class RemoveParameter : BaseDatabaseModelRefactoring
{

    /// <summary>
    /// Sets the name of the stored procedure and refactoring.
    /// </summary>
    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spRemoveParameter";
        RefactoringName = "Remove Parameter";
    }
}

/// <summary>
/// Detects whether the Rename Field refactoring has been applied between two versions of
source code.
/// </summary>
public class RenameField : BaseDatabaseModelRefactoring
{

    protected override void SetStoredProcedureAndRefactoringName()
    {
        StoredProcedure = "spRenameField";
        RefactoringName = "RenameField";
    }
}

/// <summary>
/// Detects whether a rename refactoring has been applied to two versions of source code
/// </summary>
public class RenameMethod : BaseDatabaseModelRefactoring
{

    protected override void SetStoredProcedureAndRefactoringName()
```

```
    {
        StoredProcedure = "spRenameMethod";
        RefactoringName = "RenameMethod";
    }
}
```

Results Interface

The Results class is responsible for recording any detected refactorings into the results database and also reporting back the full lists of results.

```
public interface IResults
{

    /// <summary>
    /// Adds the refactoring result.
    /// </summary>
    /// <param name="refactoringResult">The refactoring result.</param>
    void AddRefactoringResult(IRefactoringResult refactoringResult, int version);

    /// <summary>
    /// Clears the results.
    /// </summary>
    void ClearResults();


    /// <summary>
    /// Gets the results.
    /// </summary>
    /// <param name="version">The version number.</param>
    /// <returns></returns>
    DataSet GetResults(int version);

    /// <summary>
    /// Gets the versions.
    /// </summary>
    /// <returns></returns>
    DataSet GetVersions();

    /// <summary>
    /// Gets the results detail.
    /// </summary>
    /// <param name="version">The version.</param>
    /// <returns></returns>
    DataSet GetResultsDetail(int version);
}
```

Results Class

The Results class is responsible for recording any detected refactorings into the results database and also reporting back the full lists of results.

```
public class Results : IResults
{
    #region IResults Members

    /// <summary>
    /// Adds the refactoring result.
    /// </summary>
    /// <param name="refactoringResult">The refactoring result.</param>
    void IResults.AddRefactoringResult(IRefactoringResult refactoringResult, int version)
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            InsertRefactoringResultIntoDatabase(database, connection,
                  transaction, refactoringResult, version);
            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
        }
        finally
        {
            transaction.Dispose();
            connection.Close();
            connection.Dispose();
        }
    }



    #endregion

    #region Data Accessors

    /// <summary>
    /// Inserts the refactoring result into the database.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <param name="refactoringResult">The refactoring result.</param>
    private void InsertRefactoringResultIntoDatabase(Database database, DbConnection
connection, DbTransaction transaction, IRefactoringResult refactoringResult, int version)
    {
        if (database == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;

        DbParameter versionParam = command.CreateParameter();
        versionParam.ParameterName = "@Version";
        versionParam.DbType = DbType.String;
        versionParam.Value = version.ToString();
        command.Parameters.Add(versionParam);
```

```
        DbParameter refactoringParam = command.CreateParameter();
        refactoringParam.ParameterName = "@Refactoring";
        refactoringParam.DbType = DbType.String;
        refactoringParam.Value = refactoringResult.Refactoring;
        command.Parameters.Add(refactoringParam);

        DbParameter locationParam = command.CreateParameter();
        locationParam.ParameterName = "@Location";
        locationParam.DbType = DbType.String;
        locationParam.Value = refactoringResult.Information;
        command.Parameters.Add(locationParam);

        command.CommandText = "spAddResult";

        database.ExecuteNonQuery(command, transaction);
    }

    /// <summary>
    /// Clearses the results from database.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    private void ClearsResultsFromDatabase(Database database, DbConnection connection,
DbTransaction transaction)
    {
        if (database == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;

        command.CommandText = "spClearResults";

        database.ExecuteNonQuery(command, transaction);
    }

    /// <summary>
    /// Gets the results from database.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <returns></returns>
    private DataSet GetResultsFromDatabase(Database database, DbConnection connection,
DbTransaction transaction, int version)
    {
        if (database == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;

        command.CommandText = "spGetResults";

        DbParameter versionParam = command.CreateParameter();
        versionParam.ParameterName = "@Version";
        versionParam.DbType = DbType.Int32;
        versionParam.Value = version;
        command.Parameters.Add(versionParam);

        return database.ExecuteDataSet(command);
    }

    /// <summary>
    /// Gets the results detail from database.
    /// </summary>
    /// <param name="database">The database.</param>
```

```
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <returns></returns>
    private DataSet GetResultsDetailFromDatabase(Database database, DbConnection connection,
DbTransaction transaction)
    {
        if (database == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;

        command.CommandText = "spGetResultsDetail";

        DbParameter versionParam = command.CreateParameter();
        versionParam.ParameterName = "@Version";
        versionParam.DbType = DbType.Int32;
        versionParam.Value = 1;
        command.Parameters.Add(versionParam);

        return database.ExecuteDataSet(command);
    }

    /// <summary>
    /// Gets the versions from database.
    /// </summary>
    /// <param name="database">The database.</param>
    /// <param name="connection">The connection.</param>
    /// <param name="transaction">The transaction.</param>
    /// <returns></returns>
    private DataSet GetVersionsFromDatabase(Database database, DbConnection connection,
DbTransaction transaction)
    {
        if (database == null)
           throw new ArgumentNullException("Database was null when calling AddClass");
        if (connection == null)
           throw new ArgumentNullException("Connection was null when calling AddClass");
        if (transaction == null)
           throw new ArgumentNullException("Transaction was null when calling AddClass");

        DbCommand command = connection.CreateCommand();
        command.CommandType = CommandType.StoredProcedure;

        command.CommandText = "spGetVersions";

        return database.ExecuteDataSet(command);
    }

    #endregion

    #region IResults Members


    /// <summary>
    /// Clears the results.
    /// </summary>
    void IResults.ClearResults()
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            ClearsResultsFromDatabase(database, connection, transaction);
            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
```

```
        }
        finally
        {
            transaction.Dispose();
            connection.Close();
            connection.Dispose();
        }
    }

    #endregion

    #region IResults Members


    /// <summary>
    /// Gets the results.
    /// </summary>
    /// <param name="version">The version number.</param>
    /// <returns></returns>
    DataSet IResults.GetResults(int version)
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            DataSet ds = GetResultsFromDatabase(database, connection, transaction, version);
            transaction.Commit();

            return ds;
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
        }
        finally
        {
            transaction.Dispose();
            connection.Close();
            connection.Dispose();
        }
    }

    #endregion

    #region IResults Members


    /// <summary>
    /// Gets the versions.
    /// </summary>
    /// <returns></returns>
    DataSet IResults.GetVersions()
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            DataSet ds = GetVersionsFromDatabase(database, connection, transaction);
            transaction.Commit();

            return ds;
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
        }
        finally
        {
```

```
            transaction.Dispose();
            connection.Close();
            connection.Dispose();
        }
    }

    /// <summary>
    /// Gets the results detail.
    /// </summary>
    /// <param name="version">The version.</param>
    /// <returns></returns>
    DataSet IResults.GetResultsDetail(int version)
    {
        Database database = DatabaseFactory.CreateDatabase();
        DbConnection connection = database.CreateConnection();
        connection.Open();
        DbTransaction transaction = connection.BeginTransaction();

        try
        {
            DataSet ds = GetResultsDetailFromDatabase(database, connection, transaction);
            transaction.Commit();

            return ds;
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            throw ex;
        }
        finally
        {
            transaction.Dispose();
            connection.Close();
            connection.Dispose();
        }
    }

    #endregion
}
```

Encapsulate Downcast

An encapsulate downcast refactoring exists if a method exists in version n + 1 also exists in version n but with a different return type.

```
CREATE PROCEDURE [dbo].[spEncapsulateDownCast]
        @Version1 INT,
        @Version2 INT
AS
BEGIN

        SET NOCOUNT ON;

    -- Encapsulate Downcast
        -- ====================
        Select classv1.[Name] AS ClassName, methodv1.[Name] as Subject
        From Method methodv1
                Left Join Class classv1 On (classv1.ClassID = methodv1.ClassID)
        Where classv1.Version = @Version1
                -- method exists in version 2 but with different return type
                AND EXISTS (Select *
                                        From Method methodv2
                                                Left Join Class classv2 On
                                                (classv2.ClassID = methodv2.ClassID)
                                        Where classv2.Version = @Version2
                                                AND classv2.[Name] = classv1.[Name]
                                                AND methodv1.[Name] = methodv2.[Name]
                                                AND methodv1.Signature
                                                        = methodv2.Signature
                                                AND methodv1.ReturnType
                                                        <> methodv2.ReturnType
                                                AND methodv1.Code = methodv2.Code)
                -- method dos not exist in version 2 but with same return type
                AND NOT EXISTS (Select *
                                                From Method methodv2
                                                        Left Join Class classv2 On
                                                        (classv2.ClassID = methodv2.ClassID)
                                                Where classv2.Version = @Version2
                                                        AND classv2.[Name] = classv1.[Name]
                                                        AND methodv1.[Name] = methodv2.[Name]
                                                        AND methodv1.Signature
                                                                = methodv2.Signature
                                                        AND methodv1.ReturnType
                                                                = methodv2.ReturnType
                                                        AND methodv1.Code = methodv2.Code)
END
```

Push Down Method

A push down method refactoring exists if a method exists in class a in version n + 1, the same method did not exist in class a in version n, the same method did exist in the parent class of class a in version n, and the same method did not exist in the parent class of class a in version n + 1.

```
CREATE PROCEDURE [dbo].[spPullDownMethod]
	@Version1 INT,
	@Version2 INT
AS
BEGIN
	SET NOCOUNT ON;

	-- Push Down Method
	-- ================
	Select classv1.[Name] AS ClassName, methodv1.[Name] as Subject
	From Method methodv1
		Left Join Class classv1 On (classv1.ClassID = methodv1.ClassID)
	Where classv1.Version = @Version1
		-- Method does not exist in version 2 in the same class
		AND NOT EXISTS (Select *
					From Method methodv2
						Left Join Class classv2 On
						(classv2.ClassID = methodv2.ClassID)
					Where classv2.Version = @Version2
						AND methodv1.[Name] = methodv2.[Name]
						AND methodv1.Signature =
							methodv2.Signature
						AND classv1.[Name] = classv2.[Name])
		-- Does exist in a class whose super class is the class from version 1
		AND EXISTS (Select *
				From Method methodv2
					Left Join Class classv2 On
					(classv2.ClassID = methodv2.ClassID)
				Where classv2.Version = @Version2
					AND methodv1.[Name] = methodv2.[Name]
					AND methodv1.Signature
						= methodv2.Signature
					AND classv2.SuperClass
						= reverse(SUBSTRING(REVERSE(
							RTRIM(classv1.[name])),
							0,
							CHARINDEX('.', reverse(
							  rtrim(classv1.[name]))))))
		-- But didn't already exist in that class in version 1
		AND NOT EXISTS (Select *
					From Method methodv2
						Left Join Class classv2 On
						 (classv2.ClassID = methodv2.ClassID)
					Where classv2.Version = @Version1
						AND methodv1.[Name] = methodv2.[Name]
						AND methodv1.Signature
							= methodv2.Signature
						AND classv2.SuperClass =
							reverse(SUBSTRING(REVERSE(
								RTRIM(classv1.[name])),
								0,
								CHARINDEX('.', reverse(
								rtrim(classv1.[name]))))))
END
```

Extract Subclass

An extract subclass refactoring exists if class a has a child class (say, class b) in version n + 1, class b does not exist in version n, at least one method from class a in version n exists in class b in version n + 1 (and is removed from class a in version n + 1).

```
CREATE PROCEDURE [dbo].[spExtractSubClass]
	@Version1 INT,
	@Version2 INT
AS
BEGIN
	SET NOCOUNT ON;

	-- Extract Subclass
	-- ================
	Select classv1.[Name] AS ClassName, classv1.[Name] AS Subject
	From Method methodv1
		Left Join Class classv1 On (classv1.ClassID = methodv1.ClassID)
	Where classv1.Version = @Version1
		-- Method exists in version 1, but not in version 2 in the same class
		AND NOT EXISTS (Select *
				From Method methodv2
					Left Join Class classv2 On
					(classv2.ClassID = methodv2.ClassID)
				Where classv2.Version = @Version2
					AND methodv1.[Name]
						= methodv2.[Name]
					AND methodv1.Signature
						= methodv2.Signature
					AND classv1.[Name] = classv2.[Name])
		-- There is a new class whose parent class is the subjects class in version1 -
contains a method from version 1 of the parent class
		AND EXISTS (Select *
				From Method methodv2
					Left Join Class classv2 On
					(classv2.ClassID = methodv2.ClassID)
				Where classv2.Version = @Version2
					AND methodv1.[Name]
						= methodv2.[Name]
					AND methodv1.Signature
						= methodv2.Signature
					AND classv2.SuperClass
						= reverse(SUBSTRING(
						REVERSE(RTRIM(classv1.[name])),
						0,
						CHARINDEX('.',
						reverse(rtrim(classv1.[name]))))
					 AND NOT EXISTS (Select *
							From class newClass
							Where newClass.[name] = classv2.name
							AND newClass.Version = @Version1)))
		-- That method did not exist in version1
		AND NOT EXISTS (Select *
				From Method methodv2
					Left Join Class classv2 On
						(classv2.ClassID = methodv2.ClassID)
				Where classv2.Version = @Version1
					AND methodv1.[Name]
						= methodv2.[Name]
					AND methodv1.Signature
						= methodv2.Signature
					AND classv2.SuperClass
						= reverse(SUBSTRING(
						REVERSE(RTRIM(classv1.[name])),
						0, CHARINDEX('.',
						reverse(rtrim(classv1.[name]))))))
```

```
END
```

Encapsulate Field

An encapsulate field refactoring exists if a field that exists in version n + 1 also exists in version n, but is defined with a higher modifier rank in version n. *Public* is the highest modifier rank, followed by *protected,* finally *private* is the lowest rank.

```
-- encapsuate field
CREATE PROCEDURE [dbo].[spEncapsulateField]
	@Version1 INT,
	@Version2 INT
AS
SET NOCOUNT ON

Select classv1.[Name] AS ClassName, fieldv1.[Name] AS Subject
From Field fieldv1
	Left Join Class classv1 ON (classv1.ClassID = fieldv1.ClassID)
Where classv1.Version = @Version1
	-- Same field exists in version 2, but with different modifer (lower rank)
	AND EXISTS (Select classv2.[Name] AS ClassName, fieldv2.[Name] AS Subject
		From Field fieldv2
			Left Join Class classv2 ON (classv2.ClassID = fieldv2.ClassID)
		Where	classv2.Version = @Version2 AND
				fieldv1.[Name] = fieldv2.[Name] AND
				fieldv1.ModifierRank > fieldv2.ModifierRank AND
				classv2.[Name] = classv1.[Name])
```

Hide Method

A hide method refactoring exists if a method that exists in version n + 1 with a *private* modifier exists in version n with a *public* modifier.

```
CREATE PROCEDURE [dbo].[spHideMethod]
	@Version1 INT,
	@Version2 INT
AS
SET NOCOUNT ON

Select classv1.[Name] AS ClassName, methodv1.[Name] AS Subject
From method methodv1
	Left Join Class classv1 ON (classv1.ClassID = methodv1.ClassID)
Where classv1.Version = @Version1
	-- Same method exists in v1 and v2, but in v2 has a lower modifier rank
	AND EXISTS (Select classv2.[Name] AS ClassName, methodv2.[Name] AS Subject
		From method methodv2
			Left Join Class classv2 ON (classv2.ClassID = methodv2.ClassID)
		Where	classv2.Version = @Version2 AND
				classv2.[Name] = classv1.[Name] AND
				methodv1.[Name] = methodv2.[Name] AND
				methodv1.[Signature] = methodv2.[Signature] AND
				methodv1.ReturnType = methodv2.ReturnType AND
				methodv1.ModifierRank > methodv2.ModifierRank)
```

Pull Up field

A pull up field refactoring exists if a field exists in version n + 1 in class a, does not exist in class a in version n, exists in at least one sub class of class a in version n, and does not exist in the same class in version n + 1.

```
CREATE PROCEDURE [dbo].[spPullUpField]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    -- Detects a pulled up field
	-- For every field, checks whether the field does not exist
	-- in the new version and a field of the same name and type does exist
	-- in the new version under the super class, but did not in the old version
	Select Fieldv1.[Name] AS Subject, Classv1.[Name] AS ClassName
	From Field Fieldv1
		Left Join Class Classv1 on (Fieldv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
		AND NOT Exists (Select *
			From Field Fieldv2
				Left Join Class Classv2 on (Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type])
		AND EXISTS		(Select *
			From Field Fieldv2
				Left Join Class Classv2 on (Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
					0,
					CHARINDEX('.', reverse(rtrim(classv2.[name])))))
					= Classv1.SuperClass
				AND Fieldv1.[Type] = Fieldv2.[Type]
		AND NOT EXISTS (Select *
				From Field Fieldv1
					Left Join Class Classv12 on
						(Fieldv1.ClassID = Classv12.ClassID)
				Where Fieldv1.[Name] = Fieldv2.[Name]
					AND Classv12.[Name] = Classv2.[Name]
					AND Fieldv1.[Type] = Fieldv2.[Type]
					AND Classv12.Version = @Version1
		))
END
```

Extract Superclass

An extract superclass refactoring exists if two classes exist in version n + 1 with the same superclass, the same two classes in version n do not have the same superclass, at least one feature of the two classes in version n are moved to the new superclass in version n + 1.

```
CREATE PROCEDURE [dbo].[spExtractSuperClass]
        @Version1 INT,
        @Version2 INT
AS
BEGIN

        SET NOCOUNT ON;

    -- Detects an extracted super class
        -- For every class that is a super class in the new version,
        -- checks that it did not exist in the old version and one of it's
        -- child classes did
        Select Classv2.[Name] AS ClassName, Classv2.[Name] AS Subject
        From Class Classv2
                LEFT JOIN Class subclassv2 ON (subclass.SuplerClass =
                                        reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
                                                0,
                                                CHARINDEX('.', reverse(rtrim(classv2.[name])))
                                                        AND subclass.Version = @Version2)
                LEFT JOIN Class subclassv1 ON (subclass.SuplerClass =
                                        reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
                                                0,
                                                CHARINDEX('.', reverse(rtrim(classv2.[name])))
                                                AND subclass.Version = @Version1)
        Where Classv2.Version = @Version2
                AND EXISTS (Select *
                                        From Class Classv22
                                        Where Classv22.SuperClass =
                                        reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
                                        0,
                                        CHARINDEX('.', reverse(rtrim(classv2.[name])))))
                                                AND Classv22.Version = @Version2)
                AND NOT EXISTS (Select *
                                                From Class Classv1
                                                Where Classv1.SuperClass =
                                                reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
                                                 0,
                                                 CHARINDEX('.', reverse(rtrim(classv2.[name])))))
                                                AND Classv1.Version = @Version1)
                AND EXISTS (Select *
                                        From Class c1
                                                Inner Join Class c2 ON
                                                (c1.[Name] = c2.[Name] and c2.Version = 2)
                                        Where c1.Version = @Version1
                                                AND c2.SuperClass =
                                                reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
                                                0,
                                                CHARINDEX('.', reverse(rtrim(classv2.[name])))))
                AND (
                EXISTS (Select *
                        From method subclassmethodv1
                                LEFT JOIN class subclassv1 on
                                        (subclassv1.ClassID = subclassmethodv1.ClassID)
                        WHERE subclassmethodv1.Version = @Version1
                        AND NOT EXISTS (Select *
                                        From method subclassmethodv2
                                                LEFT JOIN class subclassv2 on
                                                (subclassv2.ClassID = subclassmethodv2.ClassID)
                                        WHERE subclassmethodv2.Version = @Version2
                                        AND subclassmethodv2.[Name] = subclassmethodv1.[Name]
                                        AND subclassmethodv2.[Signature]
                                                = subclassmethodv1.[Signature]
```

```
                        AND subclassmethodv2.ReturnType = subclassmethodv1.ReturnType)
                    AND EXISTS (Select *
                                From method superclassmethodv2
                                LEFT JOIN class superclassv2 on
                                (superclassv2.ClassID = superclassmethodv2.ClassID)
                                WHERE superclassmethodv2.Version = @Version2
                                AND superclassmethodv2.[Name] = subclassmethodv1.[Name]
                    AND superclassmethodv2.[Signature] = subclassmethodv1.[Signature]
                    AND superclassmethodv2.ReturnType = subclassmethodv1.ReturnType))
                    OR (
                    EXISTS (Select *
                    From field subclassfieldv1
                        LEFT JOIN class subclassv1 on
                            (subclassv1.ClassID = subclassfieldv1.ClassID)
                    WHERE subclassfieldv1.Version = @Version1
                    AND NOT EXISTS (Select *
                                From field subclassfieldv2
                                    LEFT JOIN class subclassv2 on
                                    (subclassv2.ClassID = subclassfieldv2.ClassID)
                                WHERE subclassfieldv2.Version = @Version2
                                AND subclassfieldv2.[Name] = subclassfieldv1.[Name]
                                AND subclassfieldv2.[Type] = subclassfieldv1.[Type])
                    AND EXISTS (Select *
                                From field superclassfieldv2
                                LEFT JOIN class superclassv2 on
                                (superclassv2.ClassID = superclassfieldv2.ClassID)
                                WHERE superclassfieldv2.Version = @Version2
                                AND superclassfieldv2.[Name] = subclassfieldv1.[Name]
                                AND superclassfieldv2.[Type] = subclassfieldv1.[Type]))
            )))
END
```

Remove Parameter

A remove parameter refactoring exists if a method exists in version n + 1 with at least one less parameter than in version n.

```
CREATE PROCEDURE [dbo].[spRemoveParameter]
	@Version1 INT,
	@Version2 INT
AS
SET NOCOUNT ON

Select classv1.[Name] AS ClassName, parameterv1.[Name] as Subject
From Parameter parameterv1
	Left Join Method methodv1 ON (methodv1.MethodID = parameterv1.MethodID)
	Left Join Class classv1 ON (classv1.ClassID = methodv1.ClassID)
Where classv1.Version = @Version1
	AND NOT EXISTS (Select parameterv2.*
			From Parameter parameterv2
				Left Join Method methodv2 ON
				(methodv2.MethodID = parameterv2.MethodID)
				Left Join Class classv2 ON (classv2.ClassID = methodv2.ClassID)
			Where classv2.Version = @Version2 AND
				parameterv2.[Name] = parameterv1.[Name] AND
				parameterv2.[Type] = parameterv2.[Type] AND
				classv2.[Name] = classv2.[Name])
	AND EXISTS (Select methodv2.*
			From Method methodv2
				Left Join Class classv2 on
				(classv2.ClassID = methodv2.ClassID)
			Where classv2.Version = @Version2 AND
				methodv2.[Name] = methodv1.[Name]
				methodv2.Code = methodv1.Code)
```

Push Down Field

A push down refactoring exists if a field exists in class a in version n, does not exist in class a in version n + 1, does not exist in a subclass of class a in version n, and does exist in at least one subclass of class a in version n + 1.

```
CREATE PROCEDURE [dbo].[spPullDownField]
	@Version1 INT,
	@Version2 INT
AS
BEGIN
	SET NOCOUNT ON;

	-- Push Down Field
	-- ================
	Select classv1.[Name] AS ClassName, Fieldv1.[Name] as Subject
	From Field Fieldv1
		Left Join Class classv1 On (classv1.ClassID = Fieldv1.ClassID)
	Where classv1.Version = @Version1
		AND NOT EXISTS (Select *
			From Field Fieldv2
				Left Join Class classv2 On
				(classv2.ClassID = Fieldv2.ClassID)
			Where classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type]
				AND classv1.[Name] = classv2.[Name])
		AND EXISTS (Select *
			From Field Fieldv2
				Left Join Class classv2 On
				(classv2.ClassID = Fieldv2.ClassID)
			Where classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type]
				AND classv2.SuperClass =
				reverse(SUBSTRING(REVERSE(RTRIM(classv1.[name])),
				0,
				CHARINDEX('.', reverse(rtrim(classv1.[name]))))))
		AND NOT EXISTS (Select *
			From Field Fieldv2
				Left Join Class classv2 On
				(classv2.ClassID = Fieldv2.ClassID)
			Where classv2.Version = @Version1
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type]
				AND classv2.SuperClass =
				reverse(SUBSTRING(REVERSE(RTRIM(classv1.[name])),
				0,
				CHARINDEX('.', reverse(rtrim(classv1.[name]))))))
END
```

Pull Up Method

A pull up method refactoring exists if a method exists in two subclasses sharing the same parent class in version n, and the method does not exist in those subclasses in version n + 1, and the method does exist in the parent class in version n + 1.

```
CREATE PROCEDURE [dbo].[spPullUpMethod]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    -- Detects a pulled up method
	-- For every method, checks whether the method does not exist
	-- in the new version and a method of the same name and signature does exist
	-- in the new version under the super class, but did not in the old version
	Select Methodv1.[Name] AS Subject, Classv1.[Name] AS ClassName
	From Method Methodv1
		Left Join Class Classv1 on (Methodv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
		AND NOT Exists (Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] = Methodv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Methodv1.Signature = Methodv2.Signature)
		AND EXISTS		(Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] = Methodv2.[Name]
				AND reverse(SUBSTRING(REVERSE(RTRIM(classv2.[name])),
				0,
				CHARINDEX('.', reverse(rtrim(classv2.[name])))))
				= Classv1.SuperClass
				AND Methodv1.Signature = Methodv2.Signature
				AND NOT EXISTS (Select *
					From Method Methodv1
						Left Join Class Classv12 on
						(Methodv1.ClassID = Classv12.ClassID)
					Where Methodv1.[Name] = Methodv2.[Name]
						AND Classv12.[Name] = Classv2.[Name]
						AND Methodv1.Signature = Methodv2.Signature
						AND Classv12.Version = @Version1
					))
END
```

Move Method

A move method refactoring exists if a method exists in class a in version n, does not exist in class b in version n, no longer exists in class a in version n + 1, and does exist in class b in version n + 1.

```
CREATE PROCEDURE [dbo].[spMoveMethod]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    -- Detects a moved Method
	-- For every method, checks that the method does not exist
	-- in the new version and a method of the same name and type does exist
	-- in the new version under a different class, but not in the old version
	Select Classv1.[Name] AS ClassName, Methodv1.[Name] AS Subject
	From Method Methodv1
		Left Join Class Classv1 on (Methodv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
		AND NOT Exists (Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] = Methodv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Methodv1.Signature = Methodv2.Signature
				AND Methodv1.Code = Methodv2.Code)
		AND EXISTS	(Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] = Methodv2.[Name]
				AND Classv1.[Name] <> Classv2.[Name]
				AND Methodv1.Signature = Methodv2.Signature
				AND Methodv1.Code = Methodv2.Code
				AND NOT EXISTS (Select *
					From Method Methodv1
						Left Join Class Classv12 on
						(Methodv1.ClassID = Classv12.ClassID)
					Where Methodv1.[Name] = Methodv2.[Name]
						AND Classv12.[Name] = Classv2.[Name]
						AND Methodv1.Signature = Methodv2.Signature
						AND Methodv1.Code = Methodv2.Code
						AND Classv12.Version = @Version1
					))
END
```

Add Parameter

An add parameter refactoring exists if a parameter exists for a method in version n + 1, but does not exist for the same method in version n.

```
CREATE PROCEDURE [dbo].[spAddParameterRefactoring]
	@Version1 INT,
	@Version2 INT
AS
SET NOCOUNT ON

Select classv2.[Name] AS ClassName, parameterv2.[Name] as Subject
From Parameter parameterv2
	Left Join Method methodv2 ON (methodv2.MethodID = parameterv2.MethodID)
	Left Join Class classv2 ON (classv2.ClassID = methodv2.ClassID)
Where classv2.Version = @Version2
	AND NOT EXISTS (Select parameterv1.*
		From Parameter parameterv1
			Left Join Method methodv1 ON
				(methodv1.MethodID = parameterv1.MethodID)
			Left Join Class classv1 ON
				(classv1.ClassID = methodv1.ClassID)
		Where classv1.Version = @Version1 AND
			parameterv2.[Name] = parameterv1.[Name] AND
			parameterv2.[Type] = parameterv2.[Type] AND
			classv1.[Name] = classv2.[Name])
	AND EXISTS (Select methodv1.*
				From Method methodv1
					Left Join Class classv1 on
					(classv1.ClassID = methodv1.ClassID)
				Where classv1.Version = @Version1 AND
					methodv1.[Name] = methodv2.[Name])
```

Move Field

A move field refactoring exists if a field exists in class a in version n, does not exist in class b in version n, no longer exists in class a in version n + 1, and does exist in class b in version n + 1.

```
CREATE PROCEDURE [dbo].[spMoveField]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    -- Detects a moved Field
	-- For every field, checks that the field does not exist
	-- in the new version and a field of the same name and type does exist
	-- in the new version under a different class, but not in the old version
	Select Classv1.[Name] AS ClassName, Fieldv1.[Name] AS Subject
	From Field Fieldv1
		Left Join Class Classv1 on (Fieldv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
		AND NOT Exists (Select *
			From Field Fieldv2
				Left Join Class Classv2 on
				(Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type])
		AND EXISTS	(Select *
			From Field Fieldv2
				Left Join Class Classv2 on
				(Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Classv1.[Name] <> Classv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type]
				AND NOT EXISTS (Select *
					From Field Fieldv1
						Left Join Class Classv12 on
						(Fieldv1.ClassID = Classv12.ClassID)
					Where Fieldv1.[Name] = Fieldv2.[Name]
						AND Classv12.[Name] = Classv2.[Name]
						AND Fieldv1.[Type] = Fieldv2.[Type]
						AND Classv12.Version = @Version1
					))
END
```

Rename Method

A rename method refactoring exists if a method exists in version n, and a method (that did not exist in version n) with the same code, parameters, and return type exists in version n + 1, but with a different name.

```
CREATE PROCEDURE [dbo].[spRenameMethod]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    Select Methodv1.[Name] AS Subject, Classv1.[Name] AS ClassName
	From Method Methodv1
		Left Join Class Classv1 on
		(Methodv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
	AND NOT Exists (Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] = Methodv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Methodv1.Signature = Methodv2.Signature
				AND Methodv1.Code = Methodv2.Code)
	AND NOT Exists (Select *
			From Method Methodv2
				Left Join Class Classv2 on
				(Methodv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Methodv1.[Name] <> Methodv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Methodv1.Signature = Methodv2.Signature
				AND Methodv1.Code = Methodv2.Code
			AND NOT Exists (Select *
			From Method Methodv1b
				Left Join Class Classv1b on
				(Methodv1b.ClassID = Classv1b.ClassID)
			Where Classv2.Version = @Version1
				AND Methodv1b.[Name] = Methodv2.[Name]
				AND Classv1b.[Name] = Classv2.[Name]
				AND Methodv1b.Signature = Methodv2.Signature
				AND Methodv1b.Code = Methodv2.Code
				))
END
```

Rename Field

A rename field refactoring exists if a field exists in version n, and a field (that did not exist in version n) with the same modifier and type exists in version n + 1, but with a different name.

```
CREATE PROCEDURE [dbo].[spRenameField]
	@Version1 INT,
	@Version2 INT
AS
BEGIN

	SET NOCOUNT ON;

    Select Fieldv1.[Name] AS Subject, Classv1.[Name] AS ClassName
	From Field Fieldv1
		Left Join Class Classv1 on (Fieldv1.ClassID = Classv1.ClassID)
	Where Classv1.Version = @Version1
		AND NOT Exists (Select *
			From Field Fieldv2
				Left Join Class Classv2 on
					(Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] = Fieldv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type])
		AND EXISTS (Select *
			From Field Fieldv2
				Left Join Class Classv2 on
					(Fieldv2.ClassID = Classv2.ClassID)
			Where Classv2.Version = @Version2
				AND Fieldv1.[Name] <> Fieldv2.[Name]
				AND Classv1.[Name] = Classv2.[Name]
				AND Fieldv1.[Type] = Fieldv2.[Type]
			 AND NOT Exists(Select *
			From Field Fieldv1b
				Left Join Class Classv1b on
					(Fieldv1b.ClassID = Classv1b.ClassID)
			Where Classv2b.Version = @Version1
				AND Fieldv1b.[Name] = Fieldv2.[Name]
				AND Classv1b.[Name] = Classv2.[Name]
				AND Fieldv1b.[Type] = Fieldv2.[Type]))

END
```

BIBLIOGRAPHY

Advani, D. H. (2006). Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum. *Proceedings of ACM Symposium on Applied Computing.* Dijon.

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Using metrics to identify design patterns in object-oriented software. *Proc. IEEE-CS Software Metrics Symp. (Metrics'98).*

Arisholm, E., & Briand, L. (2006). Predicting fault-prone components in a Java legacy system. *ACM/IEEE Intl. Symp. Empirical Soft. Eng.*, pp.8-17.

Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., & Di Penta, M. (n.d.). An empirical study on the evolution of design patterns. *ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007*, (pp. pp. 385-394).

Aversano, L., Cerulo, L., & Penta, M. (2009). The Relationship between Design Patterns Defects and Crosscutting Concern Scattering Degree: an Empirical Study,. *IET Software, 3*(5).

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., & Kontogiannis, K. (n.d.). Advanced Clone Analysis to Support Object-Oriented System Refactoring. *7th Working Conference on Reverse Engineering. WCRE'2000*, (pp. 98-107).

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

Beck, K. (2003). *Test Driven Development: By Example.* Addison-Wesley.

Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

Beck, K., & Gamma, E. (9981). *Test infected: Programmers love writing tests, Java Report 3(7).*

Bieman, J., & Zhao, J. (1995). Reuse through inheritance: A quantitative study of C++ software. *ACM Symposium on Software Reuse*, (pp. pp47-52). Seattle.

Bieman, J., Jain, D., & Yang, H. (2001). Design patterns, design structure, and program changes: an industrial case study. *Proceedings. IEEE Conf. on Software Maintenance (ICSM 2001)*, (pp. 580-). Florence.

Bieman, J., Straw, G., Wang, H., Willard Munger, P., & Alexander, R. (2003). Design Patterns and Change Proneness: An Examination of Five Evolving Systems. *Proc Int Software Metrics Symposium (Metrics 2003).*

Bishop, J. (2008). *C# 3.0 Design Patterns.* O'Reilly Media, Inc.

Briand, L., Daly, J., & Wust, J. (1997). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering Journal - ESE, 3*(1), pp65-117.

Briand, L., Daly, J., & Wust, J. (1999). A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering, 25*(1), pp91-121.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley.

Cain, J., & McCrindle, R. (2002). An investigation into the effects of code coupling on team dynamics and productivity. *Computer Software and Applications Conference, 2002 (COMPSAC 2002) Proceedings. 26th Annual International*, (pp. 907-913).

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

Chidamber, S., & Kemerer, C. (1994, June). A Metrics Suite for Object Oriented Design. *IEEE Transaction on Software Engineering, 20*(6), pp467-493.

Counsell, S., Hassoun, Y., & Najjar, R. (2006). 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.*

Counsell, S., Hassoun, Y., Johnson, R., Mannock, K., & Mendes, E. (2003). Trends in Java code changes: the key identification of refactorings. *ACM 2nd International Conference on the Principles and Practice of Programming in Java,.* Kilkenny.

Counsell, S., Hassoun, Y., Johnson, R., Mannock, K., & Mendes, E. (n.d.). Trends in Java code changes: the key to identification of refactorings? *Proceeding PPPJ '03 Proceedings of the 2nd international conference on Principles and practice of programming in Java.*

Counsell, S., Swift, S., & Mendes, E. (2002). Comprehension of object-oriented software cohesion: the empirical quagmire. *10th IEEE International Workshop on Program Comprehension.*

Daly, J. B. (1996). An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. ,. *Empirical Software Engineering-An International Journal, 1*(2), pp109-132.

De Lucia, A., Deufemia, V., Gravino, C., & Risi, M. (2009). Improving Behavioral Design Pattern Detection through Model Checking. *Journal of Sys. and Software, 82*(7), pp1177-1193.

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2000). Finding refactorings via change metrics. *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, (pp. pp166-177). Minneapolis.

Di Penta, M., Cerulo, L., Guéhéneuc, Y., & Antoniol, G. (2008). An empirical study of the relationships between design pattern roles and class change proneness. *International Conference on Software Maintenance*, (pp. 217-216). Beijing.

Dinh-Trong, T., & Bieman, J. (2004). Open Source Software Development: A Case Study of FreeBSD. *Proc. 10th IEEE Int. Symp on Software Metrics*, (pp. pp96-105). Chicago.

Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring — Improving Coupling and Cohesion of Existing Code. *11th Working Conference on Reverse Engineering (WCRE 2004)*, (pp. pp144-151).

Dudziak, T., & Wloka, J. (2002, Feb). Tool-supported discovery and refactoring of structural weaknesses in Code. *MS Thesis Technical University of Berlin*.

EasyMock. (n.d.). Retrieved from http://easymock.org/

Elish, K., & Alshayeb, M. (2009). Investigating the Effect of Refactoring on Software Testing Effort. *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, (pp. pp29-34).

Fenton, N., & Pfleeger, S. (1997). *Software Metrics - A Rigorous and Practical Approach Second Edition.* Int. Thompson Computer Press, London,.

Field, A. (2005). *Discovering Statistics Using SPSS.* Sage Publications.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models.* Addison-Wesley.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.*

Galli, M., Lanza, M., & Nierstrasz, O. (2005). Towards a Taxonomy of Unit Tests. *Proceedings of ESUG Research Track.*

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

Gatrell, M., & Counsell, S. (2009). Empirical Support for Two Refactoring Studies Using Commercial C# Software. *13th International Conference on Evaluation & Assessment in Software Engineering.* Durham.

Gatrell, M., & Counsell, S. (2010, September). Size, Inheritance, Change and Fault-proneness in C# software. *The Journal of Object Technology, 9*(5), pp29-54.

Gatrell, M., & Counsell, S. (2011). Design Patterns and Fault-Proneness: A Study of Commercial C# Software. *Fifth IEEE International Conference on Research Challenges in Information Science.* Guadeloupe.

Gatrell, M., & Counsell, S. (2011). Faults and their Relationship to Implemented Patterns, Coupling and Cohesion in Commercial C# Software. *International Journal of Information System Modeling and Design*.

Gatrell, M., & Counsell, S. (2012). The Effect of Refactoring on Change and Fault-Proneness in Commercial C# Software. *(Submitted to) Computer Programming Journal*.

Gatrell, M., Counsell, S., & Hall, T. (2009). Design patterns and change proneness: a replication using proprietary C# software. *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'09).* Lille.

Girba, T., Lanza, M., & Ducasse, S. (2005). Characterizing the Evolution of Class Hierarchies. *Ninth European Conf. on Software Maintenance and Reengineering*, (pp. 2-11). Manchester.

Gorschek, T., Tempero, E., & Angelis, L. (2010). A large-scale empirical study of practitioners' use of object-oriented concepts,. *Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering*, (pp. 115-124). Cape Town.

Grand, M. (2002). *Patterns in Java: A catalog of reusable design patterns ilustrated with UML, 2nd Edition.* John Wiley and Sons.

Gyimóthy, T., Ferenc, R., & Siket, I. (2005). Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Software Engineering., 31*(10), pp897-910.

Hamill, M., & Goseva-Popstojanova, K. (2009, August). Common Trends in Software Fault and Failure Data. *IEEE Transactions on Software Engineering, 35*(4).

Harrison, R., Counsell, S., & Nithi, R. (2000, June). Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. *Journal of Systems and Software, 52*(2-3), pp. 173-179.

Higo, Y., Matsumoto, Y., Kusumoto, S., & Inoue, K. (2008). Refactoring Effect Estimation Based on Complexity Metrics. *19th Australian Conference on Software Engineering, 2008. ASWEC 2008*, (pp. pp219-228).

Hitz, M., & Montazeri , B. (1996, April). Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Transaction on Software Engineering, 22*(4).

JMock.org. (n.d.). Retrieved from http://www.jmock.org/

Johnson, R., & Opdyke, W. (1993). Refactoring and aggregation. *Object Technologies for Advanced Software, irst Japan Society for Software Science and Technology (JSSST) International Symposium, 742*, 264-278.

JUnit.org. (n.d.). *JUnit.org*. Retrieved from JUnit.org: www.junit.org

Kemerer, C., & Slaughter, S. (1999). Need for more Longitudinal Studies of Software Maintenance. *Empirical Soft Engineering: An Intl. Journal, 2*(2), pp109-118.

Kemerer, C., & Slaughter, S. (1999). Need for more Longitudinal Studies of Software Maintenance,. *Empirical Soft Engineering: An Intl. Journal*, pp109-118.

Kerievsky, J. (2004). *Refactoring to Patterns.* Addison Wesley.

Kitchenham, B., Al-Kilidar, H., Ali Babar, M., Berry, M., Cox, K., Keung, J., et al. (2008). Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering, 13*(1), 97-121.

Kramer, C., & Prechelt, L. (1996). Design recovery by automated search for structural design patterns in object oriented software. *Proc. Working Conf. on Reverse Engineering*, (pp. pp208-215).

Larman, C. (2002). *Applying UML and Patterns: An Introduction to OO Analysis and Design and the United Process Second Edition.* Prentice-Hall.

Li, W., & Henry, S. (1993). First International Maintenance Metrics for the Object Oriented Paradigm., (pp. 52-60).

Mantyla, M. (2003). Bad smells in software - a taxonomy and empirial study.

Mens, T., & Tourwe, T. (2004, Feb). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30*(2), 126-139.

Mens, T., & van Deursen, A. (2003). Refactoring: Emerging Trends and Open Problems. *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE).*

Meszaros, G. (2008). *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley.

Mikhajlov, L., & Sekerinski, E. (1999). A Study of the Fragile Base Class Problem. In *Lecture Notes in Computer Science,* (Vol. 1709, p. 721).

Munro, M. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. *Proceedings of the 11th IEEE International Software Metrics Symposium.*

Murphy-Hill, E., & Black, A. (2008). Breaking the barriers to successful refactoring. *ICSE '08. ACM/IEEE 30th International Conference on Software Engineering, 2008*, (pp. pp421-430).

Murphy-Hill, E., Parnin, C., & Black, A. (2009). How we refactor, and how we know it. *IEEE 31st International Conference on Software Engineering ICSE 2009*, (pp. pp287-297).

Nagappan, N., & Ball, T. (2005). Static Analysis Tools as Early Indicators or Pre-Release Defect Density. *Proceedings of the International Conference on Software Engineering.* St Louis, US.

Najjar, R., Counsell, S., & Loizou, G. (2005). Proc. Intl. Conference on Soft. Systems Eng. and its Applications. *Encapsulation and the vagaries of a simple refactoring: an empirical study.* Paris.

Nasseri, E., Counsell, S., & Shepperd, M. (2008). An Empirical Study of Evolution of inheritance in Java OSS. *Proc. of: 19th Australian Software Eng. Conference*, (pp. pp. 269-278). Perth.

NMock. (n.d.). Retrieved from http://nmock.org/

NUnit.org. (n.d.). *NUnit.org*. Retrieved from NUnit.org: www.nunit.org

O'Cinneide, M., & Nixon, P. (1998). Composite Refactorings for Java Programs. *Proceedings of the Workshop on Formal Techniques for Java Programs. ECOOP Workshops 1998.*

Olague, H., Etzkorn, L., Gholston, S., & Quattkebaum, S. (2007, June). Empirical Validation of Three Software Metrics Suites to Predict fault-Proneness of Object Orientated

Classes Developed using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*.

Opdyke, W. (1992). *Refactoring Object-Orientated Frameworks.* Thesis.

Ostrand, T., Weyuker, E., & Bell, R. (2005, March). Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering, 31*(4).

Prechelt, L. U. (2003). A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software, 65(2)*, 115-126.

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

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley.

Sinha, V., Sinha, S., & Rao, S. (2010). Buginnings: Identifying the origins of a bug. *Proceedings of the 3rd India Software Engineering Conference*, (pp. pp3-12). Mysore, India.

Siniaalto, M., & Abrahamsson, P. (2007). A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. *Proceeding ESEM '07 Proceedings of the First International Symposium on Empirical Software Engineering and Measurement.*

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).*, (pp. 38-45). Portland.

Soares, G., Gheyi, R., & Massoni, T. (2010, July-August). Making Program Refactoring Safer. *IEEE Software, 27*(4), pp52-57.

Stroggylos, K., & Spinellis, D. (2007). Refactoring - Does It Improve Software Quality? *Fifth International Workshop on Software Quality, 2007. WoSQ'07 ICSE Workshops 2007*, (p. 10).

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

Tsantalis, N., Chatzigeorgiou, A., & Stephanides, G. (2005, July). Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering, 31*(7), pp601-614.

Usha, K., Poonguzhali, N., & Kavitha, E. (2009). A quantitative approach for evaluating the effectiveness of refactoring in software development process. *Proceeding of International Conference on Methods and Models in Computer Science, 2009. ICM2CS 2009.*, (pp. pp1-7).

van Deursen, A., & Moonen, L. (2002). The Video Store Revisited - Thoughts on Refactoring and Testing. *Proceedings of the third International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP 2002.* Sardinia.

van Deursen, A., Moonen, L., van den Bergh, A., & Kok, G. (n.d.). Refactoring Test Code. In *Extreme Programming Perspectives.*

van Rompaey, B., Du Bois, B., Demeyer, S., & Rieger, M. (2007, Dec). On the Detection of Test Smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering, 33*(12), pp800-817.

Vokac, M. (2004). Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering, 30*(12), pp904-917.

Wheeldon, R., & Counsell, S. (n.d.). Power Law Distributions in Class Relationships. *IEEE Source Code Analysis and Manipulation (SCAM)*, (pp. pp45-54). Amsterdam.

Wood, M. D. (1999). Multi-method research: An empirical investigation of object-oriented technology. *The Journal of Systems & Software,, 48(1)*, pp. 13-26.

Zaidman, A. V. (2008). Mining Software Repositories to Study Co-Evolution of Production & Test Code. *International Conference on Software Testing (ICST)*, (pp. 220-229). Lillehammer.

Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., & Baker, P. (2006). Refactoring and Metrics for TTCN-3 Test Suites. *5th Workshop on System Analysis and Modelling (SAM)*, (pp. pp148-165). Kaiserslautern, Germany.

Zhao, L., & Hayes, J. (2006). Predicting Classes in Need of Refactoring: An Application of Static Metrics. *2nd International PROMISE Workshop (co-located with the IEEE Conference on Software Maintenance).* Philadelphia, Pennsylvania USA.