Declaration of Authorship

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

Rajaa Najjar

Date: 10 September 2008

An Empirical Study on Encapsulation and Refactoring in the Object-Oriented Paradigm

by

Rajaa Najjar

A Thesis Submitted in Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

in the

University of London

September, 2008

School of Computer Science and Information Systems

Birkbeck, University of London

Abstract

Encapsulation lies at the heart of the Object-Oriented (OO) paradigm by regulating access to classes through the private, protected and public declaration mechanisms. Refactoring is a relatively new software engineering technique that is used to improve the internal structure of software, without necessarily affecting its external behaviour, and embraces the OO paradigm, including encapsulation.

In this Thesis, we empirically study encapsulation trends in five C++ and five Java systems from a refactoring perspective. These systems emanate from a variety of application domains and are used as a testbed for our investigations. Two types of refactoring related to encapsulation are investigated. The 'Encapsulate Field' refactoring which changes a declaration of an attribute from public to private, thus guarding the field from being accessed directly from outside, and the 'Replace Multiple Constructors with Creation Methods' refactoring; the latter is employed to remove code 'bloat' around constructors, improve encapsulation of constructors and improve class comprehension through the conversion of constructors to normal methods. Both types of refactoring have a strong bond with the need for proper encapsulation of classes and objects as well as with other OO constructs such as inheritance. Overall results demonstrate both quantitative and qualitative benefits in terms of code removal, improved encapsulation and better understanding of system traits in both C++ and Java.

In the name of God, the Most Merciful the Most Compassionate

Acknowledgements

In writing this Thesis, there have, of course, been many people who have inspired, guided and supported me in various ways and so deserve mention and credit. In particular, I am very grateful to my supervisor Professor George Loizou not only for his very insightful and encouraging comments and suggestions on the Thesis, but also for our long discussions at the beginning of the project right through the end.

I also owe a great debt of gratitude to my supervisor Dr Steve Counsell for his great support, patience, and guidance that helped shape and inform my knowledge and understanding of this material. Both George and Steve remain for me an incomparable model of scholarly diligence and generosity. I cannot begin to express how much I have learned from them and how much I continue to learn.

I would also like to thank Professor Martin Shepperd and Dr Tracy Hall for their constructive suggestions. I also owe special thanks to Dr Peter Sozou at the London School of Economics for his statistical expertise, and to Phil Gregg and the staff at Birkbeck College and all my friends for their support and assistance.

My greatest thanks go to my husband Mataz for his support and to my daughter Sarah who is still too young to read this work, but made life such a pleasure despite its moments of frustration. It is to these two people that this Thesis is dedicated, with thanks, appreciation and love.

To my Mum and Dad, brothers and sisters, I say I am very grateful for your moral support and sincere prayers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANA	Average Number of Ancestors
BSF	Bean Scripting Framework
CBO	Coupling Between Objects (Chidamber and Kemerer metric)
DIT	Depth of Inheritance Tree (Chidamber and Kemerer metric)
DAM	Data Access Metric
EF	'Encapsulate Field' refactoring
GNU	Gnu's Not Unix
LCOM	Lack of Cohesion in Methods (Chidamber and Kemerer metric)
LEDA	Library of Efficient Data Algorithms
LOC	Lines Of Code
MOOD	Metrics for Object-Oriented Design
NOC	Number Of Children (Chidamber and Kemerer metric)
OO	Object-Orientated
OOP	Object-Oriented Programming
RFC	Response For a Class (Chidamber and Kemerer metric)
RMCCM	'Replace Multiple Constructors with Creation Methods' refactoring
RM	'Rename Method' refactoring
SE	Software Engineering
UML	Unified Modelling Language
WMC	Weighted Methods per Class (Chidamber and Kemerer metric)

CHAPTER 1 Introduction

1.1 Introduction

The Object-Oriented (OO) paradigm was first proposed in the early 1990's even though its underlying principles have been around for considerably longer (Parnas 1972, Liskov et al. 1977). Nowadays, the OO paradigm has become the most widely used approach to problem-solving, with languages such as C++ and Java dominating the commercial IT world. These languages have established such a foothold, that it is unlikely we will see a paradigm shift and a wholesale move to the use of any other languages in the near future. It is of immense importance that we therefore try to understand best practice in the use of these languages by developers. To place this importance in context, many legacy systems exist whose maintenance burden accounts for a large part of developer time and cost. It is a well-known truism that maintenance accounts for approximately 70% of software development costs (Pressman 2000). Proper adoption of the OO paradigm is one means by which we can address the issue of future maintenance and, some would say, learn the lessons of mistakes made in the past.

The OO paradigm incorporates a number of important features such as encapsulation and inheritance (Stroustrup 1991). From an *encapsulation* perspective, the OO paradigm implements private, protected and public access specifiers as part of its syntax. Encapsulation was incorporated into programming languages to provide the developer with a means by which access to the private features of a class could be controlled (or encapsulated) and public features made available to all classes. The responsibility for enforcing these access mechanisms was thus devolved to the compiler and as a result the developer was freed from associated implementation and testing issues.

For example, we can define an attribute of a Java class as private and know that only methods of that class can access that attribute. Equally, if we want to allow a method to be

accessed from outside the class in which it is defined, we can declare that method to be public. In C++, different rules apply; the *friends* facility therein (Stroustrup 1991) allows access by classes to the private features of another class, thereby subverting the principle of encapsulation. The use of friends stores up future maintenance difficulties, since we can no longer rely on the compiler to prevent programming side-effects. Frequent misuse of friends in this sense, together with other inappropriate use of encapsulation, thus leads to a spiral of poor maintenance, anomalies and 'decay' in code (Counsell and Newson 2000).

The concept of *inheritance* is also a salient feature of OO languages because it promotes the reuse of code and, in theory, reflects the way that we as humans structure and manipulate information. Proper use of inheritance requires the careful consideration of how class features should be encapsulated. While public features can be shared by any other class, the *protected* keyword relates specifically to the use of inheritance by allowing, for example, only subclasses of a class X to access the protected class features of X.

A strong and symbiotic bond therefore exists between encapsulation and inheritance; appropriate use of one feature requires careful consideration of the other. Many past and ongoing empirical studies have attempted to capture the essential characteristics of these features to inform our understanding of OO software.

While anecdotal evidence suggests that inheritance is an aid to program understanding, there is conflicting evidence in the research literature (Daly et al. 1996, Cartwright and Shepperd 2000). It is not entirely clear that the use of inheritance does deliver the stated benefits and there is evidence to suggest that, in certain circumstances, the use of flat systems (containing no inheritance) may be more appropriate. Snyder (Snyder 1986) suggests that the introduction of inheritance compromises the benefits that proper encapsulation brings; the complexities associated with understanding different levels of classes may be a factor in this ongoing debate.

In turn, as a software engineering (SE) community, we know very little about the role that encapsulation plays in its links with inheritance or how developers maintain encapsulation

principles in systems. A few studies have examined encapsulation trends in OO software (Snyder 1986, Briand et al. 1999b, Skoglund 2003, Schärli et al. 2004).

One SE technique that has grown significantly in importance in recent years is that of refactoring (Fowler 2000, Kerievsky 2004). Refactoring refers to a technique whereby changes are made to a program to improve its design without necessarily changing the semantics of that program. As well as a better program design, the benefits of refactoring include improved program understandability and, in theory, improved maintainability of that program. In the seminal text on refactoring by Fowler (Fowler 2000), refactoring is stated as being the reversal of code decay and thus should be applied at every stage of a program's life. In the same text, the mechanics of 72 different Java refactorings are described, and, for each, the circumstances as to when the refactoring under consideration should be applied are also stated and justified. Many of the 72 refactorings relate specifically to an inheritance hierarchy. For example, the 'Extract Superclass' refactoring creates a superclass from existing classes (appropriate encapsulation mechanisms are required as part of the mechanics). Equally, many refactorings involve, for example, the explicit use of encapsulation. The 'Encapsulate Field' refactoring changes the declaration of an attribute from public to private, thereby taking advantage of Java encapsulation. Consideration of the effect of this refactoring on any subclasses or superclasses is required as part of the mechanics of that refactoring.

While the OO features of encapsulation and inheritance have a strong relationship, when cast in terms of a refactoring context, their relationship becomes even more pronounced. The purpose of this Thesis is thus to examine trends from an empirical point of view according to these three inter-related perspectives, namely, encapsulation, inheritance and refactoring.

In view of the above, we empirically investigated the patterns in declarations of attributes and methods in the classes of five C++ and five Java systems, with specific recourse to where those classes were placed in the inheritance hierarchy. Some quite revealing results relating to violation of encapsulation were found; the role that the C++ 'friends' facility played in this violation was remarkable. The results of this initial study informed the next

logical step in the research – to explore the potential for the application of relevant refactorings amongst the 72 proposed by Fowler. As a result, an empirical study of the 'Encapsulate Field' refactoring (Fowler 2000) amongst the Java systems studied revealed significant potential for the application of this refactoring.

During our investigation of the systems studied, a trend that appeared frequently was the proliferation of class constructors in many of the classes. The problem with large numbers of class constructors is that they invite code 'bloat', i.e., too much unnecessary and duplicated code in the class (Kerievsky 2004). A further and spiralling problem then results; developers will ignore existing constructors in favour of their own declarations. The fact that constructors are usually defined as public but need not be, if appropriate steps are taken, meant there was significant potential for the application of other refactorings.

In this Thesis a further empirical study was undertaken which employed the 'Replace Multiple Constructors with Creation Methods' refactoring (Kerievsky 2004). The study demonstrated that savings in lines of duplicated code and comment lines could be obtained from application of the refactoring (through the process of removing 'code bloat').

The three empirical studies combined illustrate how trends in software systems can be uncovered and analysed.

1.2 Motivation

The motivation for the work in this Thesis stems from the following drivers:

Firstly, to understand the encapsulation features prevalent in C++ and Java systems. The SE community knows little about trends in encapsulation, how those trends have arisen and even less about how they impact the shape of inheritance hierarchies in systems, and the problems of maintaining those systems.

Secondly, the concept of refactoring is well understood from a practical perspective and the benefits it has to offer are significant. Yet, the SE community still knows little about the potential for application in quantifiable and qualitative terms of applying refactoring; only limited evidence has thus far been produced (Tokuda and Batory 2001, Counsell et al. 2006).

1.3 Objectives and Contribution

The objectives of this Thesis are:

1. To obtain a greater understanding of C++ and Java systems from an encapsulation perspective; this is complemented by the study of how inheritance figures in those encapsulation trends for each type of system considered.
2. To assess the quantitative and qualitative benefits of applying refactorings related specifically to the concepts of encapsulation and inheritance in Java systems. In particular, to investigate those refactorings where both encapsulation and inheritance have a strong influence.

Software metrics (Shepperd and Ince 1993, Fenton and Pfleeger 2002) will be the vehicle for the analysis of our empirical studies, taking care to consider their direct relevance and applicability for the goals set in each of those empirical studies.

The Thesis makes a number of contributions to SE from an empirical perspective and these contributions have been published in a number of archived sources.

The Thesis has contributed significantly to highlighting problems that arise and trends that emerge in software systems as they evolve. In particular, the tendency for anomalies to occur from an encapsulation perspective in the systems studied. This is likely to have been the result, or cause, of knock-on effects in other parts of the systems, with other OO constructs becoming embroiled.

It does additionally contribute to an understanding of the benefits that can be gleaned from applying different refactorings, as well as some of the problems and trade-offs that have to be considered before undertaking any refactoring. More empirical studies need to be undertaken in this sense to build up a body of knowledge in this industrially important area (Baker et al. 2006, Zeiss et al. 2006) of SE; we see our contribution as a step in that direction.

Finally, we feel careful description and analysis of the problems of data collection and methodological aspects of undertaking research using empirical studies informs the choice of techniques in future empirical studies. It is as important to learn from the process by which empirical studies are undertaken as it is to draw conclusions from their outcomes.

The Thesis therefore contributes to two strands of research. Firstly, few previous studies have empirically investigated the role that encapsulation plays and the extent to which it is used or abused in OO systems, yet encapsulation is a fundamental part of the OO paradigm. The Thesis makes a contribution to our understanding and knowledge in this area, see Chapter 4. Equally, few studies have investigated the empirical interplay and practical relationships between encapsulation and inheritance (another fundamental aspect of OO) and the trends that systems exhibit as a result. The Thesis contributes to our understanding and knowledge about those relationships and the interplay thereof, again see Chapter 4.

Secondly, from a refactoring perspective, few studies have empirically explored the potential (both qualitative and quantitative) for applying refactorings in which encapsulation and inheritance play a central part. The Thesis contributes to our understanding of the empirical opportunities, pitfalls and practicalities of undertaking such refactorings, see Chapters 5 and 6. Underpinning both strands are issues related to data collection, hypothesis setting and appropriate statistical analyses.

1.4 Application Domains

The two sets of systems that we use as a testbed throughout the Thesis represent a wide spread of application domains and considerable care was taken to ensure that the two sets of systems were comparable in terms of the application domains they addressed. Two libraries, a compiler, a framework and a graph editor were chosen; from available C++ systems and five Java systems chosen also. This spread will allow conclusions to be drawn not only across the two languages, but between application domains as well. The reason behind choosing these two languages among many other OO languages is that the C++ is the forerunner of Java (C++ evolved into Java).

The five C++ systems are: Edge, Rocket, ET++, GNU and LEDA. The five Java systems are: GraphDraw, BSF, Libjava, Barat and Swing. More details on these ten systems are available in Chapter 3.

1.5 A Research Framework

An important feature of the research described in the Thesis is the combination of recognised threads of the OO paradigm (i.e., encapsulation and inheritance) *and* emerging SE disciplines, in our case refactoring. In this Thesis, we draw on the theoretical underpinnings of both to inform our practical investigations; we also recognise that the motivation for applying each draws from many other SE disciplines. For example, there is a strong tie between refactoring and software testing, since we are required to test after each step of a refactoring. Equally, a fundamental reason for applying a refactoring or set of refactorings is to make software easier to comprehend and maintain. Reengineering software has strong ties with refactoring, and economic factors such as cost and effort also play a large part in dictating the extent to which these activities can be undertaken. SE is not a discipline with discrete un-connected elements – it consists of inter-connected elements which work together. Figure 1.1 illustrates some of the key SE concepts and how these are related to refactoring.

At a more detailed level, the mechanics of each individual refactoring draw on the fundamental elements of OO – that of a class and its constituent elements. Methods and attributes are also subject to the rules of language syntax and semantics. A refactoring may be simple and low-level requiring minor changes to code or complex and high-level embracing many other refactorings.

Figure 1.2 shows the key elements that constitute the core of the Thesis from the interplay between encapsulation, inheritance and refactoring. It shows the relationships embodied by the Encapsulate Field (EF) and Replace Multiple Constructors with Creation Methods (RMCCM) refactorings, both central to the Thesis contents.

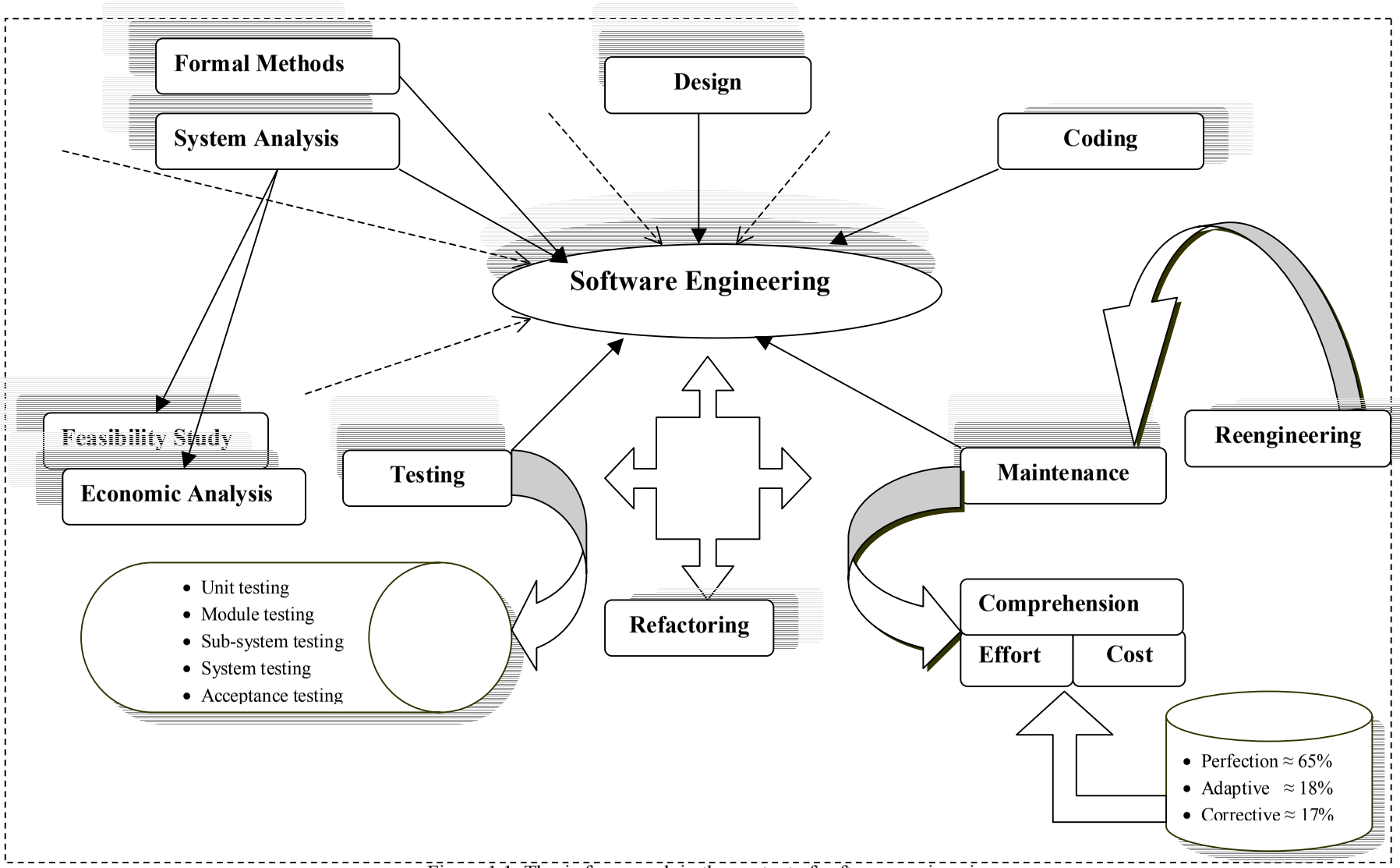

Figure 1.1: Thesis framework in the context of software engineering

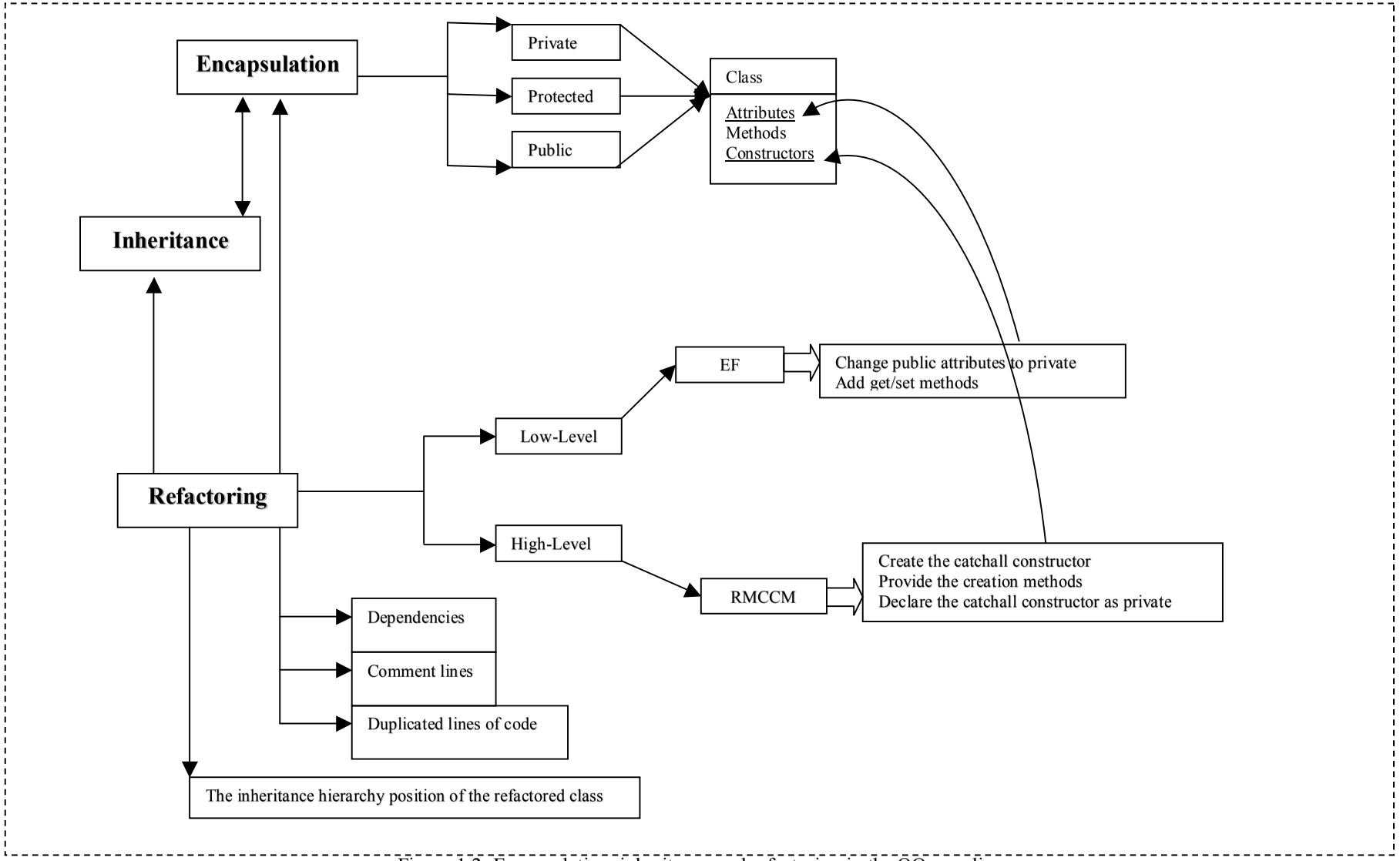

Figure 1.2: Encapsulation, inheritance and refactoring in the OO paradigm

1.6 Overview of the Thesis

This chapter presents the context and motivation of our work, and gives an overview of our objectives and contributions. The five C++ systems and the five Java systems referred to earlier are used as a testbed for our empirical investigations throughout the Thesis.

Chapter 2 describes related work to the research problems addressed. It looks at related and complementary work in the area of empirical SE, software metrics, refactoring, design patterns (Gamma et al. 1995) and antipatterns (Brown et al. 1998, Laplante and Neill 2006). It also provides insights and justification for the nature of the research presented in this Thesis.

Chapter 3 provides a detailed description of the research methodology adopted in the Thesis including the basis upon which the systems used in the study were chosen, justification for the choice of statistical analysis used (together with detailed supporting analysis), the assumptions on which the Thesis rests and the process by which research decisions were made. As a result of the strong emphasis in the Thesis on data collection (and its associated vagaries), we also provide a preliminary study comparing the manual and automatic collection of data for the five Java systems presented in Chapter 3.

Chapter 4 gives a description of an empirical study in which trends of encapsulation and inheritance, from the five C++ systems and the other five Java systems, described in Chapter 3, were investigated. Here, the trends in the use of class features declared as private, protected and public for inheriting and non-inheriting classes were investigated. Results showed that declaring attributes as public, which violates the principle of encapsulation, appears to exist in most of the systems investigated.

Chapter 5 presents an empirical investigation in which problems, opportunities and benefits associated with applying the 'Encapsulate Field' refactoring (Fowler 2000) were examined. A sample of classes from each of the five Java systems was chosen and the EF refactoring was then considered. EF is one of the most encapsulated-related refactorings proposed to date. Results indicated several key reasons why this particular refactoring can be either

simple or difficult to implement in practice, depending on the features of the class and its role in the inheritance hierarchy.

Chapter 6 empirically investigates again, the problems, opportunities and benefits of refactoring class constructors across a sample of classes from the five Java systems. The 'Replace Multiple Constructors with Creation Methods' refactoring (Kerievsky 2004) was applied to each of a set of classes containing three or more constructors (see Section 3.3.4). Results showed benefits in terms of removed (duplicated) lines of code across the majority of systems; they also showed the potential for improved class comprehension by the creation of non-constructor methods and improved encapsulation of class elements through the use of a private catchall constructor.

Finally, Chapter 7 provides the conclusions and contributions of the research presented in this Thesis with reflection on the original objectives and the extent to which they were achieved. It also gives some insights into related future research.

CHAPTER 2 A Survey of Related Work

2.1 Introduction

In the previous chapter we gave an introduction to the Thesis and described the contents of each chapter. In this chapter, we describe work related to the research presented in this Thesis. This includes a description of work in the fields of empirical SE, software measurement and metrics. It also describes relevant work on refactoring, design patterns and antipatterns. For each of these areas, we examine related and complementary work, justifying why and in which aspects our work is different.

In Section 2.2 we discuss the theme of empirical studies in SE related specifically to the OO paradigm. In Section 2.3 we describe related work in the use and analysis of OO metrics. In Section 2.4 we give an in-depth presentation on encapsulation and inheritance. In Section 2.5 we present a historical synopsis on refactoring, as a new method of improving software design, and we provide justification for choosing and examining empirically the types of refactoring undertaken in the context of the Thesis. In Section 2.6 we provide a briefing in the area of patterns and antipatterns, and their relationship with our work, specifically refactoring.

2.2 Empirical Software Engineering

SE is still considered to be a young engineering discipline that needs much more empirical research to build up its body of knowledge so as to correspond with those of other engineering disciplines. Empirical SE can be defined as an approach to SE designed to assess the strengths and weaknesses of various software products, processes, and resources at their (different) developing stages. We can thus characterise, evaluate, control,

understand and predict such software-related entities. In this respect, empirical studies are as important for SE as for any other engineering discipline.

Much work has been done to find out the problems associated with conducting an empirical study and to support empirical SE research. A number of these works have presented useful recommendations, guidelines, and suggested steps for conducting an empirical study (Seaman 1999, Briand and Wust 2002, Kitchenham et al. 2002). Not surprisingly, most of them have suggested that in order to achieve proper improvement in empirical SE, there must be more concentration on replicated studies that can be conducted on a large number of environments and the results compared. One of the key motivations in our work is to carry out an empirical investigation with different types of applications and different programming languages to reinforce our knowledge about specific OO issues.

Lewis et al. (Lewis et al. 1991) carried out a controlled experiment designed to evaluate the impact of the OO paradigm on software reuse when compared to the procedural paradigm. The OO characteristics of encapsulation, inheritance and data abstraction were considered key elements for software reuse. They found that the OO paradigm improved software productivity partially because of the effect of software reuse. The empirical study thus confirmed the hypothesis that the OO paradigm supports reusability more easily than the procedural paradigm.

An early empirical study conducted by Basili et al. (Basili et al. 1986) presented a framework for analyzing a variety of experimental work performed over several years. The aim of the framework, as a mechanism, was to facilitate the definition, planning, operation, and interpretation of past and future studies in order to learn lessons, and learn further from identifying problems in such experiments. The application of such a framework is important for getting the most out of an experiment and facilitating the opportunity for replicating experiments especially using the same definitions and operations. For our work we can state: the *motivation* is to understand, assess, learn and improve our knowledge about how developers deal with the concept of OO *encapsulation*; the *objects* are metrics and theory, the *purpose* is to characterise and evaluate, the *perspective* is the researcher, the

domain is program/project and the *scope* is a multi-project variation study to identify OO encapsulation anomalies that are to be ameliorated via refactoring.

Briand and Wust (Briand and Wust 2002) found that many of the measures in the literature were redundant, i.e., the number of metrics was much higher than the captured dimensions. It also appeared that the metrics suite proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994) received the most attention in empirical studies. In other words, *coupling*, *inheritance* and other *size-related metrics* have been intensively investigated. In terms of our research this provides us with the opportunity to consider empirically investigating other OO features not previously investigated thoroughly, such as that of encapsulation referred to above.

Seaman (Seaman 1999) presented a study on qualitative methods in empirical studies of SE. The work claimed that recent empirical studies in SE have been recognised in the SE community for addressing the human role in software development. It was argued that qualitative methods and quantitative methods could be adapted and incorporated into the design of empirical studies in SE in order to take advantage of the strengths of both qualitative and quantitative methods; qualitative analysis in SE research is less widely used than quantitative analysis. Also, any assessment based on the latter without metrics is no more than an opinion. In a study by Perry et al. (Perry et al. 2000), the strengths and weaknesses of empirical research were presented. A general structure for software empirical studies and concrete steps for achieving a number of goals were described. These are: designing better studies, collecting data more effectively and involving others in empirical enterprises. The authors believed that in order to improve the contemporary state of empirical research, better studies had to be carried out and more credible conclusions and interpretations drawn from them. They stated that:

"*... unless we understand the specific factors that cause tools and methods to be more or less cost-effective, the development and use of particular technology will essentially be a random act. Empirical studies are a key way to get this information and move towards well-founded decisions*".

The main step in improving our understandability about software techniques, methods and tools, therefore, is to carry out well-structured empirical studies and draw reliable conclusions from them, thus contributing, to some extent, to the body of knowledge in empirical SE.

Kitchenham et al. (Kitchenham et al. 2002) reported guidelines necessary for all relevant kinds of empirical work and addressed the needs of different stakeholders (i.e., researchers and practitioners). They also pointed out that in many SE experiments, the selected design is complex, and the analysis method is inappropriate for coping with it.

In addition, they indicated that as part of *data collection guidelines,* data collection is problematic in empirical software studies because software measures are not well-defined. They stated that the purpose of data collection guidelines is to ensure that the data collection process is defined well enough for experiments to be replicated. They also found that some problems in SE studies are related to *unstandarised* software metrics. In Chapter 3 we investigate empirically the extent to which manual data collection differs from the automatic one from the perspective of the reliability of the former versus the latter; this was necessitated by the fact that data collection for some systems could only be done manually.

A survey and description of the major, recent empirical studies of OO artifacts, methods and processes was carried out by Briand et al. (Briand et al. 1999a). They presented a number of factors that needed to be taken into account for successful empirical studies. Once again, the need for replication studies was pointed out as a key factor to successful empirical research. Their work was the impetus for our research to empirically investigate *encapsulation trends* in two different OO languages (C++ and Java) and with four different types of application. This investigation is comprehensively presented in Chapter 4.

Moreover, the introduction of new techniques to support an OO software development process, such as *refactoring,* places a burden on all stakeholders to propose new metrics and to thoroughly investigate such techniques empirically. Consequently, the need for empirical studies and the lack of such studies in certain areas of OO SE also inspired us to carry out an empirical study focused on OO *encapsulation* mechanisms and further

empirical investigations of the opportunities, problems and benefits of encapsulation-related refactorings across a number of Java software systems. Chapters 5 and 6 deal extensively with the EF and RMCCM refactorings, respectively.

2.3 Software Metrics

Software metrics are 'tools' to measure software quality. For almost forty years, researchers and software practitioners have been active in the area of SE measurement. Tom Gilb (Gilb 1976) was believed to be the first person who used the term 'software metrics'. Tom DeMarco (DeMarco 1982) strongly supports the need for measurement in software development: "*You can neither predict nor control what you cannot measure*". According to Fenton and Pfleeger (Fenton and Pfleeger 2002, p. 28) *measurement* is a mapping of empirical objects to numerical objects with preservation of all relations and structures.

In the SE field, metrics are used to assess software quality early in the software development process in order to make changes that might reduce complexity and improve the long-term viability of the end-product (Hall et al. 2005). In a sense, software metrics can support decision-making during the software life cycle. In theory, a distinction between *direct* and *indirect* measurement of an attribute should be made (Fenton and Pfleeger 2002, p. 39). A direct measurement of an attribute of an object does not depend on any other attributes of the same or other objects. For example, class size can be measured in terms of its total number of method instances. Indirect measurement involves one or more other attributes; for example, module defect density can be measured in terms of the *number of defects* and *module* size measures. We also need to distinguish between *internal* and *external attributes* of a software artifact or process; internal attributes are those attributes which can be measured merely in terms of the artifacts or processes themselves, while *external attributes* of an artifact or process are those attributes which can only be measured with respect to how the artifact or process relates to its environment (Fenton and Pfleeger 2002, p. 74). For example, internal attributes of a software code can be size, reuse and coupling, and the external ones can be usability, maintainability, and reliability.

In the past, a large number of software metrics have been proposed by researchers and practitioners for analysing software systems. In the early stages, metrics were proposed for measuring software, based on the structural paradigm, such as: cyclomatic complexity (McCabe 1976), fan-in and fan-out (Henry and Kafura 1981) and lines of code (Rosenberg 1997). With the introduction of OO technology, many measures have been proposed to take into account the new technology and to analyse the quality of OO software at both theoretical and empirical levels.

Much work has been done on proposing and investigating, theoretically and empirically, metrics for OO software (Abreu and Carapuca 1994, Chidamber and Kemerer 1994, Lorenz and Kidd 1994, Bieman and Zhao 1995, Harrison et al. 1998b, Briand et al. 1999b, Cartwright and Shepperd 2000). In (Harrison et al. 1998b) for example, a set of OO metrics, called MOOD, were investigated in respect of measurement theory and validated empirically by using three different application domains. They concluded that the MOOD set of metrics could be used to provide an overall software quality assessment of the systems studied. They also found that the MOOD metrics can work complementarily with the Chidamber and Kemerer metrics (Chidamber and Kemerer 1994); MOOD can provide assessment for the software at system level, whereas the Chidamber and Kemerer metrics can provide assessment at class level.

A number of studies have also tried to show how, in practice, OO systems are not exhibiting the features we expected they would. The research described in (El Emam et al. 2001) is one such example, using a C++ telecommunications *framework* as a basis of the study. Currently, empirical studies seem to be shifting from proposing new software metrics into investigating the properties and applications of available software metrics, and further replicating such investigations (Prechelt et al. 2003, Kanmani et al. 2004, Bocco et al. 2005).

An early text on OO software metrics is that by Lorenz and Kidd (Lorenz and Kidd 1994). They proposed eleven OO design metrics and provided ‘rules of thumb’ for some of the metrics (here we understand the term rules of thumb to mean broad guidelines on optimal sizes for each of the metrics). They further divided OO metrics into four categories, i.e.,

size, inheritance, internals and externals. A number of design metrics to measure class size in different ways were defined, such as the *number of public methods in a class,* considered to be a good measure of the amount of responsibility in the class. Moreover, the *number of variables in a class,* which counts all the private and protected variables, in addition to the *public* variables if there are any, defined in a class, is also a measure of class size.

Since software metrics provide the means by which software can be measured and systems compared with each other, a commonly used metric in the OO paradigm is the Depth in the Inheritance Tree (DIT) metric, originally proposed by Chidamber and Kemerer, indicating how far down in the inheritance hierarchy (level 0 is considered the base class) a class is. Some empirical studies have shown that the deeper a class in the inheritance hierarchy is, the more difficult that class will be to understand (Basili et al. 1996). Theory suggests that in order to understand a class deep down in an inheritance hierarchy, every class above that class has to be understood as well; any modification of a class in an inheritance hierarchy may cause side-effects for other classes. The study described in (Cartwright and Shepperd 2000) also showed that classes with higher change densities were found to be in the lowest level of inheritance structures. This may be attributed to the fact that developers tend to extend or add more functionality to classes at the lower level of inheritance structures when probably there is lack of time, or simply because it is easier to do so. At the same time, this draws attention to the need for more empirical studies to accept or refute such claims with the support of software metrics. Other empirical studies have shown that flat systems (containing no inheritance) are actually easier to understand than systems containing inheritance (Harrison et al. 2000), or the deeper the inheritance hierarchy for a software system is the more difficult it has become to be maintained (Daly et al. 1996). In (Daly 1996) it was found that the most common reason among developers for the difficulty in understanding C++ software was inheritance. That is to say, if the design was inappropriate, inheritance would be a barrier to our understanding of software.

In our view supported by our empirical results (see Section 4.5) and as suggested by other studies (Snyder 1986, Skoglund 2003, Schärli et al. 2004), encapsulation and inheritance are two interrelated concepts and should be studied in parallel.

In (Berard 1995), five characteristics that can lead to specialised metrics were defined. These characteristics were: *Localisation*, *Encapsulation*, *Information Hiding*, *Inheritance* and *Abstraction* (Kramer 2007). Interestingly, the study considered encapsulation and information hiding as two elements in support of measurement at a higher level of abstraction and which provide a good indication of the quality of OO design. Ten years before, Snyder (Snyder 1986) studied the relationship between encapsulation and inheritance. It was suggested then that the introduction of inheritance severely compromised the benefits encapsulation could offer.

It is important that any metrics used should be validated theoretically and empirically; thus they must represent the attributes they qualify. Validation is therefore essential to the success of software metrics (Shepperd 1995). Fenton and Pfleeger (Fenton and Pfleeger 2002) state a definition for validating a software measure as:

"*The process of ensuring that the measure is a proper numerical characterisation of the claimed attribute by showing that the representation condition is satisfied*".

A framework for software measurement validation was presented in (Kitchenham et al. 1995). They identified concepts that are necessary for measurement: entities (real-world objects), attribute (an entity property), units (determines how to measure an attribute) and scale type (nominal, ordinal, interval or ratio). Such a framework can help researchers and practitioners focus on what they mean by entities and attributes and how to define and measure them more consistently.

Finally, the role that human influence plays in the development process needs to be considered. A number of studies have investigated the factors that contribute to project success or otherwise (Hall et al. 2005); appropriate use of collected metrics plays a key role in the assessment and discovery of those factors. In terms of setting up a metrics programme, significant thought needs to be given to its purpose and the factors that will contribute to its success (Hall and Fenton 1997).

Throughout the Thesis we have consistently followed and strictly adhered to the above-mentioned validity criteria for all measures employed.

2.4 Encapsulation

The OO paradigm is characterised by a number of salient features; for example, encapsulation, inheritance and polymorphism. Encapsulation, though not originating from the OO paradigm, is considered as a class structure, where internal class elements are separated from external ones. In other words, most of the class data (attributes) should be private and the methods that operate on them public. Encapsulation can minimise the side-effects of changes to a system when those changes take place. It also facilitates component reuse, and reduces overall system coupling (Briand et al. 1999b, Pressman 2000), and may also improve the *understandability* of a system.

Firstly, it is important to point out that encapsulation and information hiding are often confused by researchers as well as practitioners. It seems that the lack of standard definitions for these two concepts led to this confusion. Encapsulation and information hiding were originally established in a structural environment (and then incorporated into the OO paradigm). Both terms seem to be different although they are often considered identical or similar in the OO community. Some believe that these two terms refer to the same thing; others consider them to be different. In OO technology, encapsulation is the creation of self-contained modules that contain both the data and the processing. Information hiding, on the other hand, has been defined, according to Parnas (Parnas 1972), a pioneer in this field, as hiding the most changeable design decisions in a module of a program in order to protect the remaining parts from further changes.

Though there are many definitions for information hiding and its connection with encapsulation, it seems that various researchers approach the subject from different perspectives (Parsons 1994, Fowler 2000, Pressman 2000, Budd 2002). In (Parsons 1994) encapsulation means linking together all data and the operations that work on that data in one unit (class), while information hiding means using encapsulation to separate the public parts of an object from its private ones, while also hiding the details of the implementation from the other units (classes). Budd (Budd 2002) refers to information hiding as hiding the implementation of the operations of a class from other parts of the system under consideration. Fowler (Fowler 2000) points to encapsulation (data hiding) as hiding class

attributes from other classes by making them private, so making no distinction between encapsulation and data hiding.

The most accepted definition for encapsulation is the one documented in (Snyder 1986). Snyder defined encapsulation in terms of defining strict external interfaces (members can be accessed from outside their class) between separately-written modules with the intention of minimising the interdependencies between these modules. Thus encapsulation means separating private members from public ones. Those researchers, who claim that these two terms, namely, encapsulation and information hiding, are different, base their view on the fact that not every encapsulated member is hidden.

From our point of view, information hiding is more concerned with hiding implementation details, and at the same time it cannot be achieved in OO without encapsulation. For example, for a class *c,* say, with *n* methods, there is no requirement for other classes in the system to know how the class *c* implements those *n* methods; at the same time the class *c* cannot hide the implementation of the *n* methods without being encapsulated in *c*. On the other hand, we believe encapsulation as an OO principle indicates two distinct facets. Firstly, it is a technique of enclosing data and operations performed on the data within one unit called a 'class'. Secondly, encapsulation is a practice that provides access control to the remaining parts of the system by specifying which class members should be private, and which ones public. Thus, both data and some of the operations acting on the data can be hidden (private members); consequently we can refer to this as information hiding. For instance, it should be possible to change the type of, or rename, an attribute of a class without affecting the classes (clients) that use such changed classes (hiding data, private member). It should also be possible to change the implementation of any method (operation) in any class without affecting its clients; for example, changing the type of one of the variables from *integer* to *double* or replacing a number of statements in the contained method with a new method. We consider information hiding as a structural concept, where encapsulation is the mechanism that provides the access visibility for class members through the use of access specifiers provided by OO language syntax. Our definitions of encapsulation and information hiding can be found in the Glossary of Terms.

Most OO languages support encapsulation. In C++ and Java, three mechanisms allow classes to control which other classes can have access to their members. These are *private, protected* and *public*. In addition, C++ also provides the *friend* mechanism, which controls the accessibility among a group of classes, not necessarily related in a single hierarchy. Public members are those that are available to all clients and form the external interface. When inheritance is considered, class members should be declared as protected to facilitate access to the required parent class members, while private members should be considered for internal use by the contained class members. Also, Java has default access which allows class members that are declared without an access specifier to have full access to all members of classes in that package excluding private ones. Table 2.1 shows the levels of accessibility for each type of access control in Java.

Specifier	Class	Package	Subclass	World
Private	Yes	No	No	No
Default (no specifier)	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Table 2.1: The access levels in Java

There is empirical evidence to suggest that encapsulation is practised improperly (Skoglund 2003). This is realised by declaring class members, specifically the class attributes, as public, so every other class can have access to those members and can manipulate them. In an experiment conducted by Skoglund (Skoglund 2003), software engineers were used and their view of encapsulation issues examined. Some of the interviewed subjects stated that, because of time pressure and testing reasons, they often changed private declarations to public. In a similar vein, an empirical investigation into the exploitation of OO features was carried out by using small-sized C++ programs developed by undergraduate students (Kanmani et al. 2004). The main goal of this research was to investigate OO metrics for measuring coupling, cohesion and inheritance at the class level. One of the observed results showed that inheritance was properly used among the programs in the design of the class level attributes and the classes were highly cohesive; however, most of the attributes and the methods of the classes were declared to be public.

The importance of encapsulation and its effects on other OO fundamental features, such as inheritance, were well recognised and considered. Snyder (Snyder 1986) studied the relationship between encapsulation and inheritance and suggests that the introduction of inheritance severely compromises the benefits encapsulation can offer. Thus, proper use of inheritance generally requires us to consider protected declarations in classes when we want to restrict access to only subclasses of a particular class. Few studies, however, made the connection between available OO metrics for coupling, cohesion, or complexity metrics, and encapsulation. In (Rosenberg and Hyatt 1997), a high value of the LCOM cohesion metric was considered as a support for class encapsulation. In the same study it was stated that: "*…effective object-oriented designs maximise cohesion since it promotes encapsulation"*. Since this metric was originally proposed by Chidamber and Kemerer (Chidamber and Kemerer 1994), they failed to give an explanation as to how encapsulation would be improved. Thus the aforesaid quoted sentence remains to be substantiated.

In the study conducted by Harrison et al. (Harrison et al. 1998b), the authors empirically evaluated and validated the MOOD set of metrics originally proposed by Abreu and Carapuca (Abreu and Carapuca 1994). They found that the six MOOD metrics were shown to be valid within the context of the framework they provided. However, in terms of the two metrics *Attribute Hiding Factor* (*AHF*) and *Method Hiding Factor* (*MHF*), Harrison et al. found that it was difficult to agree that AHF and MHF could be used as indirect measures of encapsulation. In other words, they considered these metrics as a measure for information hiding. Therefore, we postulate that providing a standard definition for encapsulation is essential for our work in this Thesis.

According to Jagdish and Davis (Jagdish and Davis 2002) there are no metrics which measure encapsulation; consequently they introduced the *Data Access Metric* (DAM), which is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. The DAM metric ranges over [0,1] and high values are considered desirable. Nevertheless, they do not measure encapsulation in terms of class method declarations and the distribution of private, protected and public methods. More recently, a study carried out by Laing and Coleman (Laing and Coleman 2006) described the results of developing an approach for formulating a set of orthogonal OO metrics. The

aim of the research was to produce a minimal set of OO metrics capable of analysing code quality with the same degree of accuracy as afforded by a metrics set of significantly larger cardinality. Interestingly, the minimal set of metrics that capture encapsulation and polymorphism was presented in terms of the total number of local methods (inherited methods are not counted) and the total number of remote methods for a class; strangely, they did not consider class attributes in their metrics set.

To summarise, most OO mechanisms have received a large amount of attention by the SE community. However, encapsulation as an OO key concept has received less attention when compared with inheritance and polymorphism. Many metrics have been proposed and validated to assess the use of inheritance, such as the DIT and the ANA; relatively few have been proposed for measuring encapsulation. Even though the earlier studies suggested that proper use of inheritance required concomitant consideration of encapsulation, researchers and practitioners alike do not seem to make a clear connection between encapsulation and inheritance. The need for metrics and empirical studies to assess the influence of encapsulation on the way we write OO systems has motivated our research. In Chapter 4 we empirically investigated the trends of encapsulation and in Chapter 5 we use the EF encapsulation-related refactoring for improving the internal structure of software design. The next section reviews refactoring and its impact on encapsulation.

2.5 Refactoring

One of the techniques widely used to improve the structure of software systems is refactoring. The term refactoring was first used by Opdyke and Johnson in their 1990 paper (Opdyke and Johnson 1990). Software restructuring can be considered as the origin of refactoring. Software restructuring according to Chikofsky et al. (Chikofsky et al. 1990) is "*the transformation from one representation form to another at the same relative abstraction level, while preserving the system's external behavior"*. Fowler (Fowler 2000) defines refactoring in two forms; stating that refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without*

changing its observable behaviour", and "*to restructure software by applying a series of refactorings without changing its observable behaviour*". Put in other words, the refactoring process can improve software design by tidying up code, moving code to the right place or removing code. Therefore, we can say that refactoring propels programmers to work more deeply on understanding what the code does and is hence an aid to maintenance and reuse (Johnson and Foote 1988).

There is a growing interest in refactoring due to its major role in supporting maintenance and reuse in OO software systems (Fowler 2000, Tokuda and Batory 2001, Counsell et al. 2003, Kerievsky 2004, Advani et al. 2005). Fowler in his seminal text (Fowler 2000) identified 72 types of refactoring. He illustrated each type of refactoring with simple examples using the notation of UML (Rumbaugh et al. 1998). In this Thesis we empirically investigated one of Fowler's refactorings, namely, the low-level EF refactoring (Fowler 2000) which is related to encapsulation (see Chapter 5). In addition, in Chapter 6 we empirically investigated the high-level RMCCM refactoring highlighted by Kerievsky (Kerievsky 2004).

The Ph.D. work of Opdyke (Opdyke 1992) described various types of refactoring applicable to OO software and proposed a technique to automate the refactoring process. He presented three types of refactoring in detail related to inheritance and aggregation. He also demonstrated how to automatically support refactoring in a way that would preserve program behaviour. Opdyke and Johnson (Opdyke and Johnson 1993) described a study in which they illustrated how to create abstract superclasses from other classes via refactoring. They decomposed the refactoring operation into a set of refactoring steps, and provided examples. They also discussed a technique that could automate these steps, thus making the process of refactoring applicable in practice. In Johnson and Opdyke (Johnson and Opdyke 1993), some common refactorings based on aggregation, including how to convert from inheritance to aggregation, and how to reorganise an aggregate/component hierarchy are reported. They also describe how to refine aggregations by moving variables and methods between aggregate and component classes, and how to move variables and methods within inheritance hierarchies. However, Johnson and Opdyke, in the two previous studies, did not provide results for applying such refactorings on a real system, as an empirical validation of

their work. In Chapters 5 and 6 we empirically investigated opportunities, benefits, and problems of the two types of refactoring referred to earlier. This work validates empirically, to some extent, the work of Fowler and Kerievsky.

Very little empirical data addresses the question of how widespread refactoring is in practice. Empirical work in the refactoring area and its automation is found in Tokuda and Batory (Tokuda and Batory 2001), where fourteen thousand lines of code were transformed automatically which would otherwise have had to be carried out by hand. In (Counsell et al. 2003) an empirical study was carried out on a set of library classes. In that paper, the *substitute algorithm* refactoring (Fowler 2000) (the substitute algorithm refactoring can be described as a modification of the body of a method to improve the way it functions) was found to be the most popular type of refactoring identified. Moreover, a survey of software refactoring was carried out by Mens and Tourwe (Mens and Tourwe 2004). They describe refactoring activities and the techniques that support such activities. For example, the activity of identifying where to apply refactoring is supported by the identification of bad smells (Fowler 2000) and the activity of the assessment of the refactoring effect on software quality is supported, for example, by software metrics and empirical measurements. Mens and Tourwe point out that the type of application domain has a great impact on identifying the type of refactoring to be applied. They also discovered that refactorings can be classified according to the quality attributes they affect. Finally, recent work by Advani et al. (Advani et al. 2005) describes the results of an empirical study of the trends across multiple versions of open source Java software. A specially developed software tool extracted data related to each of fifteen refactorings from multiple versions of seven Java systems according to specific criteria. Herein, we have carried out a similar study and manually examined the applicable classes for three types of refactoring; EF, chain constructors and RMCCM (Najjar et al. 2003, Najjar et al. 2005).

One of the interesting areas in refactoring is the natural connection between refactoring and design patterns presented in the seminal textbook by Gamma et al. (Gamma et al. 1995). In Kerievsky's text (Kerievsky 2004) too, the link between refactoring and design patterns was established. Kerievsky describes, in a well structured way, how developers can introduce and remove patterns from code. As part of our work, we empirically investigated

the high-level refactoring of RMCCM, on Java software systems, in view of the fact that encapsulation can be improved through the creation of private objects (Najjar et al. 2003). More details can be found in Chapter 6 of this Thesis.

2.6 Patterns and Antipatterns

Design patterns (Gamma et al. 1995) are becoming increasingly popular as a way of describing solutions to general design problems. A controlled experiment was carried out by Prechelt et al. (Prechelt et al. 2001), in which they investigated whether exploiting design patterns in software design was useful or harmful. An example was presented where using design patterns made a program harder to maintain. However, they emphasised, due to the unexpected new requirements, that it was preferable to use design patterns in software construction. An analogous study, conducted by Bieman et al. (Bieman et al. 2001), empirically investigated the relationships between class size, inheritance and design patterns. Their conclusion, in terms of design patterns, was that classes that played roles in design patterns were more change-prone than others.

Moreover, Bieman et al. (Bieman et al. 2003) replicated the previous study on the system studied in (Bieman et al. 2001) and four additional systems each of which was implemented in Java. The results of the replication study were found to support previous work in four out of the five case studies; they accounted for change-proneness of pattern-participant classes as being the classes that provide key functionality to the four systems. These studies reflect how much work needs to be done, so that developers can effectively use design patterns to improve OO design. As part of our research we empirically investigated the RMCCM refactoring of Kerievsky (Kerievsky 2004), which is based on the *factory method* pattern (Gamma et al. 1995) (see Chapter 6). The results show that quantitative and qualitative benefits can be gained by applying this refactoring.

Brown et al. (Brown et al. 1998) stated that patterns can often evolve into antipatterns, as shown in Figure 2.1. They defined antipatterns as commonly used solutions to problems which in turn produce negative consequences; the aim of identifying and studying

antipatterns is to describe forms that can then be the subject of refactoring effort. Since both design patterns and antipatterns deal with solutions, the difference between them is in the context; an antipattern is a pattern used inappropriately thereby generating negative consequences. Refactoring is used to evolve the available solution to a better one by improving its structure. So, identifying software antipatterns can inform our knowledge about refactoring, and thus help to understand the internal structure of OO software design.

The focus of this Thesis is on empirical studies; this, therefore, requires the collection of data. How this data is collected, and how representative it can be, is of paramount importance. The next chapter deals, to the extent required, with this topic. It also deals with the statistical techniques employed in our empirical studies as well as the software systems studied and why they were chosen.

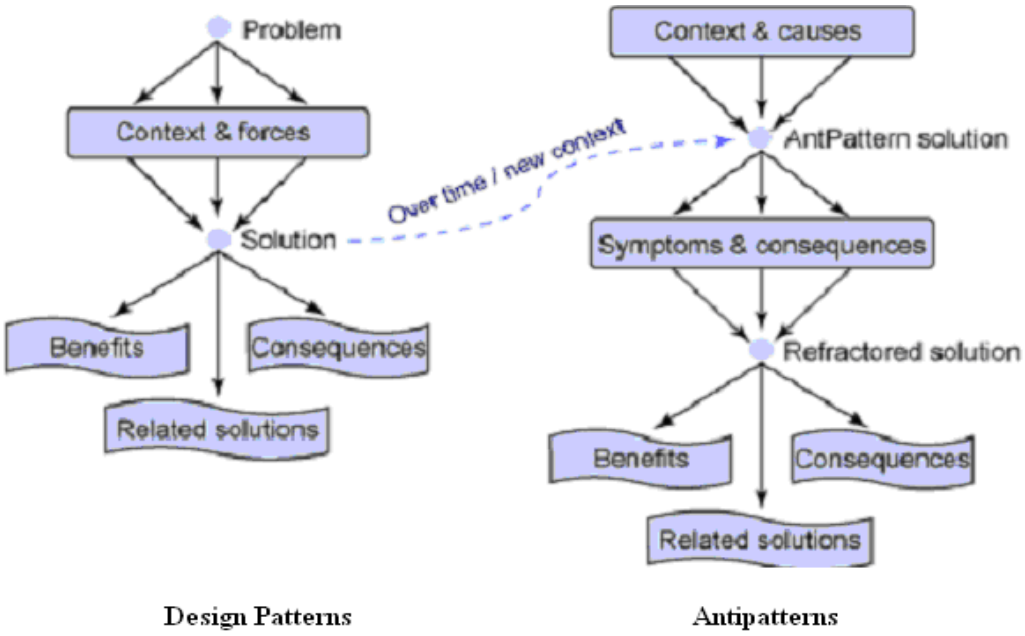

Figure 2.1: Patterns versus antipatterns (Source:(Brown et al. 1998))

CHAPTER 3 Research Methodology

In Chapter 2 related work was surveyed and critiqued, and where appropriate, how the research presented in this Thesis fits within the overall context of that chapter. In this chapter we describe our research strategy and how we make our decisions about the main issue of the empirical research. Since our empirical studies depend on data collected sometimes automatically and on other occasions manually, we also found it necessary and useful to investigate the difference between automatic and manual data collection.

3.1 Introduction

The benefit of doing research is a critical question for any researcher. Research in a particular area of SE provides information that can help (software) managers making informed decisions to deal successfully with (software) problems. The information provided could be the result of a careful analysis of data collected by the researcher using one of the data collection techniques, such as surveys, questionnaires, interviews, experiments and project artifacts. Collected data can be qualitative or quantitative depending on the nature of the research questions and the research methodology adopted. The technique that a researcher adopts to discover the answers to his or her research questions is called 'the research methodology'. Two schools of thought are recognised in terms of research methodology; quantitative and qualitative. The choice of which to adopt affects data collection, data analysis and discussions of validity.

This chapter consists mainly of two parts: the first part deals with research methodology adopted, and the second part is concerned with the comparison between manual and automatic data collection. We note that the preliminary research on which the second part of this chapter is based was first published in (Najjar et al. 2004).

Section 3.2 gives the description of research methods in empirical SE. Details of the research design presented in this Thesis and the decisions for conducting the empirical investigations are described in Section 3.3 and its subsections. Section 3.4 describes an empirical study of the difference between automatic and manual data collection.

3.2 Research Methods in Software Engineering

Empirical studies in SE can be performed quantitatively, qualitatively, or in combination. The qualitative approach deals with text, pictures and can be used to understand human behaviour and reaction towards a situation (such as levels of understanding and communication) (Seaman 1999). It also involves a personal subjective interpretation of the data. On the other hand, the quantitative approach compares numerical data (Perry et al. 2000). There is no one single preferable approach for conducting a research. However, it could be argued that the approach that serves the purpose of getting most of the required information for the research problem(s) and finding out the solution(s) is preferable (qualitative, quantitative, or a combination thereof).

The research presented in this Thesis is quantitative in nature; it uses software metrics collected from ten C++ and Java software systems and pertaining to encapsulation, inheritance and refactoring. Encapsulation is one of the fundamental characteristics of OOP. It helps separate responsibilities and provides an easy way to understand software. A well-encapsulated class needs less time and effort to understand in order to make changes. The class in this sense is more maintainable due to the communication between the class and its clients through its public interface. In such a class, data fields cannot be directly changed from outside because of their private declarations. Consequently, fewer faults are introduced because the clients of the class cannot inadvertently make any changes to the data fields without knowledge of the class. With the introduction of inheritance, developers should consider the *protected* declaration for the inherited features. Encapsulation and inheritance could be thought of as joint mechanisms. Developers have to decide which features are restricted only to the inherited classes, and declare them *protected*. However, inappropriate declarations can be corrected at any stage of the class life (it is preferable that

this is done as soon as the class is created); using refactoring techniques to achieve this improves class encapsulation.

In the next section we describe details on how and why decisions were made to empirically investigate encapsulation and inheritance on the basis of the selected C++ and Java software systems. Furthermore, we consider whether refactoring could be applied to these systems from the perspectives of encapsulation and inheritance.

3.3 Research Design

Making plans for any project is one of the key elements of the project success. In terms of research, setting the limits and strategy can help researchers decrease the amount of work required to complete the research and then draw conclusions.

In general, empirical investigations can be performed using one of the main techniques: *case studies*, *experiments, surveys* and others. Surveys are retrospective and there is no control on the study variables. Such studies are performed if an organisation wants to determine how a population feels about a particular set of issues. Case studies and experiments, on the other hand, are not retrospective. Researchers have more control on their variables. The decision to carry out a case study or experiment depends on the level of control required on the variables. To put it another way, carrying out an experiment requires researchers to have a high level of control on the variables. Moreover, experiments can be used to support generalisation of the research findings, whereas surveys and case studies can be used to confirm findings about a tool or a method on a single organisation (Fenton and Pfleeger 2002). This research adopts an artifact archived analysis, a technique which is different from surveys, case studies, and experiments. Our approach is based on investigating software artifacts, mainly, the source code of ten software systems described in Section 3.3.3.2. This will aid our understanding of how developers write software system code, specifically in terms of encapsulation, inheritance, and encapsulation-related refactorings. In other words, the study aims to investigate how developers put theory into practice from the perspectives of encapsulation and inheritance.

A counter-argument may suggest that an approach based on experiments help researchers to further generalise their research findings because a formal experiment is carefully controlled and contrasts different values of the controlled variables; its results are generally applicable to wider community and across many organisations (Fenton and Pfleeger 2002). We believe that using software artifacts is the right way for tackling the research problem referred to above. Since refactoring is a new technique for improving OO software systems, identifying and exploring the opportunities, benefits, and problems of applying refactoring can be considered as a first step in applying this new research area. Thus, after having known all the problems associated with refactoring, experiments can be carried out to support or refute the findings of such studies. The following sections describe the research strategy followed in order to achieve the research objectives and deliver the contributions.

3.3.1 Identifying research objectives

After describing the research problem, it is essential to establish the research direction. This is accomplished by identifying the research objectives that describe why we want to do this particular research and what we want to accomplish.

In this Thesis, as presented in Chapter 1, we empirically investigate encapsulation, inheritance and encapsulation- and inheritance-related refactorings. The motivation and objectives of the research are presented in Chapter 1, where we describe how these objectives are covered in the remaining chapters and how they are presented and followed throughout the Thesis.

This chapter investigates the role that errors play in manual data collection. This is in order to establish and provide an insight into the quality of manual data collection (there are many occasions when manual data collection is inevitable).

In Chapter 4 we investigate the evolution of the C++ and Java software systems from encapsulation and inheritance perspectives. In Chapter 5 we investigate the potential, opportunities and benefits of applying EF refactoring. Finally, in Chapter 6 we investigate the opportunities and benefits of applying chain constructors and RMCCM refactorings.

3.3.2 Generating hypotheses

To investigate the research objectives it is important to generate the hypotheses that describe and interpret these objectives. A list of hypotheses pertaining to manual data collection, encapsulation and inheritance are described in this chapter and Chapters 4 and 6.

The hypotheses are formulated formally by describing and stating the *null* and the *alternative* hypotheses. The null hypothesis often refers to no (significant) relationship/difference between two variables. The alternative hypothesis, on the other hand, refers to the existence of the relationship or the difference between the variables. In other words, the alternative hypothesis represents the research question and forms the researcher's prediction, while the null hypothesis states the opposite prediction of what the researcher thinks (Field 2006). Moreover, the alternative hypothesis can be directional or non-directional (Field 2006). It is always assumed that the null hypothesis is true unless the data reveals the opposite. Therefore, testing the null hypothesis is the starting point of accepting or rejecting the alternative hypothesis.

3.3.3 Identifying testbed software systems

One of the challenges that researchers encounter when carrying out empirical studies in any scientific discipline is choosing the appropriate sampling method. There are two major types of sampling design: probability and non-probability samplings. The main difference between these two types of sampling is that the non-probability sampling does not involve *random* selection while probability sampling does. That is, the probability of selecting any element from the population, using probability sampling, is known, and the elements, of non-probability sampling do not have a known probability of being selected as sample subjects. Moreover, generalisation is one of the key aims that every researcher is keen to satisfy when selecting the research sample. However, there are many obstacles in defining a study sample, such as identifying the population elements, getting all the elements and selecting from them a representative sample. Most importantly is accessing those members. In SE, identifying all the software systems that are based on the OO paradigm is almost impossible. Moreover, there are access restrictions to some categories of these systems and, as such, it is hard to carry out ideal probability sampling.

3.3.3.1 Sampling techniques and criteria of choosing the software systems

In this Thesis, a non-probability, *purposive* sampling technique is used to classify research samples. One important reason for choosing non-probability over probability sampling is cost and time; this is because software systems are often not available from a single source and not all of them can be accessed by researchers. Non-probability purposive sampling is used to identify the ten software systems according to the following criteria: five software systems are implemented in C++ and five in Java; the software systems are real, not toy systems, differ in their sizes (in terms of the number of classes) and belong to a wide range of application domains (a library package, compiler, graph-editor, and framework). Some of these systems are from well established software system corporations (such as IBM, GUN and Sun), and have also gone through different maintenance iterations. The diversity of application domains and diversity of sizes help identify important common trends or differences in terms of those three areas across these applications. Moreover, having relatively few representative samples leads to a greater depth of information from the carefully selected software systems.

The methodological trade-off between similarity and variety of application domain of the selected samples is to achieve variation-based generalization. In other words, the ten software systems could have been based on *one* type of application domain only; however, our decision is justified on the basis that we want to obtain more in-depth information about OSS across *different* application domains. As mentioned earlier, time and cost are two key elements that play a role in determining which and how many systems can be investigated. Twenty or more systems could have been investigated in this research but that was not possible due to time limit pressure and the viability of investigating one research problem for a long period of time. The choice of the ten systems provides a reasonable basis for comparison of different application domains and between the two languages (Java and C++).

Replication is another important factor in terms of being able to generalise our research findings. In this respect, we can say that the research presented in this Thesis (and the analysis of the systems) can be replicated, further supporting the case for generalising the findings of the Thesis.

A *stratified* sampling technique is used in Chapter 5 to extract a set of classes from each of the five Java systems investigated. This method is chosen to ensure that the sample includes an adequate number of all sets of classes that can have the same number of attributes. The procedure for extracting the sample of classes from each of the five Java systems was identical. The set of classes with at least two attributes was ranked according to the number of attributes and every *fifth* class was then selected. The reason behind excluding classes with only one attribute is that, after reviewing these classes, we found that most of these attributes were private or protected, and as such EF refactoring could not be implemented. Since the Java systems we are investigating have different sizes, in terms of the number of classes, (GraphDraw has 49 classes, Swing 1043, for example), we chose to take every fifth class, after arranging the classes in each of the five Java systems in ascending order (in terms of the total number of attributes each class contains) and grouping them into a set whose elements are sublists consisting of five classes in ascending order. The last sublist may contain less than five classes; this occurs when the total number of classes in a system is not divisible by 5. Thus the sample contains the tail of each sublist. This stratified sampling (Coolican 1990) yielded a wide range in the size of classes for our empirical analysis.

3.3.3.2 Software systems description

The following describes the ten software systems (C++ and Java) used as a testbed basis for the work presented in this Thesis:

The five C++ systems are:

1. System One: **Edge**, a graph editor, consisting of approximately 30.8 thousand non-comment source lines (KNCSL). The number of classes available in the header files is 41.

2. System Two: **ET++**, a user interface framework, consisting of approximately 56.3 KNCSL. 205 classes are available in the header files.

3. System Three: **Rocket**, a compiler, consisting of 32.4 KNCSL. The number of classes available in the header files is 209.

4. System Four: **GNU**, the C++ class library, consisting of 53.3 KNCSL. The number of classes in the header files is 103.
5. System Five: **Library of Efficient Data Algorithms (LEDA)** consisting of about 123 KNCSL and 203 classes available in the header files.

The required data for GNU and LEDA was collected manually. This was due to the unavailability of the source code in an online format. In Table 4.1 these numbers of classes are referred to as 'Sample Size'

The five Java systems are:

6. System Six: **GraphDraw**, a Java graph-editor VGJ (Visualizing Graphs with Java), is a tool for graph drawing and graph layout. Graphs can be input into VGJ in two ways: with a textual description (GML), or through a drawing the user creates using a graph editor. The number of files with the extension ".java" is 52, containing 52 classes.

7. System Seven: **Bean Scripting Framework (BSF)**, an architecture for incorporating scripting into Java applications and applets. Scripting languages such as Netscape Rhino (Javascript), VBScript, Perl, Tcl, Python, NetRexx and Rexx are commonly used to augment an application's function or to script together a set of application components to form an application. The number of files with the extension ".java" is 60, containing 65 classes.

8. System Eight: **Libjava**, the language sub-library set of 89 Java classes available from the public domain at the Gnu gcc website. The number of files with the extension ".java" is 83.

9. System Nine: **Barat**, a compiler front-end for Java. Barat is a framework that supports static analysis of Java programs. It parses Java source code files and class

files and builds a complete abstract syntax tree from Java source code files, enriched with name and type analysis information. The number of files with the extension ".java" is 369, containing 407 classes.

10. System Ten: **Swing Java Package Library**, which provides a set of 'lightweight' (all-Java language) components that, to the maximum degree possible, perform the same on all platforms. The number of files with the extension ".java" is 506, containing 1248 classes.

The following table summaries the classes of the Java systems considered by class category.

System	Total number of (.java) files	Total number of classes	Number of Interfaces	Number of abstract classes	Number of concrete classes
GraphDraw	52	52	2	1	49
BSF	60	65	5	1	59
Libjava	83	89	4	4	81
Barat	369	407	155	31	221
Swing	506	1248	96	109	1043

Table 3.1: Breakdown of the number of interfaces, abstract classes and concrete classes found in each of the five Java systems

3.3.4 The chosen refactorings

Refactoring should theoretically improve maintainability. Maintainability can be defined as the ability to modify or make changes to the existing functionality of a software system. One of the goals of good maintainability is to undertake maintenance without impacting other components of the system. It is also known that software maintainers spend 70% of the software cost on maintaining the software (Pressman 2000) and 60% out of maintenance cost is expended on understanding software code. Therefore, writing code with maintainability in mind helps, to some extent, to reduce the long term maintenance expenses.

Refactoring is a technique for improving the design of existing code and reversing code decay (Fowler 2000), it is meant to improve code understandability; theoretically, it should help improve code maintainability. In theory, refactoring software systems makes them easier to understand and modify. In this Thesis we investigate opportunities, benefits and problems associated with two main types of refactoring related specifically to encapsulation

and inheritance. Encapsulate Field (EF) as a low-level refactoring (i.e., refactoring that can be done on the basis of adding, deleting or modifying a class member) (Fowler 2000), and Replace Multiple Constructors with Creation Methods (RMCCM) as a high-level refactoring (i.e., refactoring that can be done on the basis of modification to hierarchies or introducing design patterns) (Kerievsky 2004). Furthermore, testing is an important underpinning to refactoring. The essential precondition for doing refactoring is to have solid tests (Fowler 2000). Robust tests can help determine if the software code has been broken after anything is changed. Refactorers, therefore, should improve the software test suite in order to ensure that doing refactoring will not break the code. So applying refactoring to software systems, theoretically, would improve maintenance and allow more time for testing software systems.

The three types of refactorings that are used primarily in this research are *EF*, *chain constructors* and *RMCCM*. EF, a low-level refactoring, is selected from a pool of 72 types of refactoring (Fowler 2000). These (72) refactorings involve simple changes at a class level such as rename, delete or move method/attribute. The EF refactoring is one of the most encapsulation-related refactorings that improves encapsulation by changing class attribute declarations from public to private.

The second type of refactoring, *chain constructors* is a pre-requisite for the third type, RMCCM refactoring. RMCCM is a high-level refactoring since it involves the use of design patterns (creation methods). The choice of these two refactorings from 27 high-level refactorings (Kerievsky 2004) is based on the expected improvement that could be provided to class encapsulation through the declaration of the catchall constructor as private, and through the declaration of the creation methods as protected in the presence of inheritance.

3.3.5 Software metrics definitions

Providing a clear definition of software metrics would help in replicating the study to support or refute the study findings (Kitchenham et al. 2002).

Herein, we provide the definition of the collected metrics that are used throughout the Thesis. Other metrics, related to each chapter, are not included in the following list.

1. ***priPA***: number of private primitive attributes.
2. ***priNPA***: number of private non-primitive attributes.
3. ***priC***: number of private constructors.
4. ***priPM***: number of private primitive methods.
5. ***priNPM***: number of private non-primitive methods.
6. ***proPA***: number of protected primitive attributes.
7. ***proNPA***: number of protected non-primitive attributes.
8. ***proC***: number of protected constructors.
9. ***proPM***: number of protected primitive methods.
10. ***proNPM***: number of protected non-primitive methods.
11. ***pubPA***: number of public primitive attributes.
12. ***pubNPA***: number of public non-primitive attributes.
13. ***pubC***: number of public constructors.
14. ***pubPM***: number of public primitive methods.
15. ***pubNPM***: number of public non-primitive methods.

3.3.5.1 Criteria of choosing the software metrics

We chose to use attributes and methods as a basis for our study since they represent the quintessential elements of an OO class. Since we are interested in investigating how developers declare class members (attributes, methods and constructors), and the possibility of changing some of these member declarations, we collected all the metrics related to the types of access modifiers for class members. The access modifiers (private, protected and public) of class members and the class location in an inheritance hierarchy are considered as the core of the metrics definitions. These metrics are simple and can be collected easily.

3.3.6 Data collection

Collecting data manually is a process fraught with difficulties. However, the author has adopted a method that helped to make this collection free of incorrect data as far as possible. At the same time there are a number of confounding factors that affected the manual data collection.

In order to keep track of the data counting, a technique was developed that helped to obtain the correct figures as far as possible. This technique was based on using colours, shapes, lines, and counting numbers. For example, all private members were coloured in green, protected in blue and public in red; attributes were underlined, and methods circled in their appropriate colour, with the associated counting number.

There are, however, a number of confounding factors that can lead to incorrect figures in terms of the number of class members. Having a large class (in terms of the number of attributes, methods and constructors, for example) will make it difficult for someone to keep track of a series of nine numbers (3 private metrics, 3 protected, and 3 public) spanning possibly forty pages or more. The fatigue issue, that is, the ability of dealing with numbers, differs from one person to another. Most people become tired and lose concentration after two/three hours. Distractions also play a role in getting incorrect figures.

There are, yet again, a number of aspects that can be considered as positive confounding factors such as learning system styles, i.e., learning how developers tend to organise class members. For example, some developers tend to declare all the attributes at the end of the class, others declare all constructors after declaring all class attributes members. Also some of the systems tend to have relatively small classes which lends itself to correct manual data collection in general.

On the other hand, the automatic collection of data is considered to represent the correct data. The results from the automatic data collection were meticulously verified. Whenever the software tool (see Appendix B) was used to collect the data based on reading tokens from the input-file, it was checked by reading every single token in that file and then writing all the read data to an output-file. The output-file was then checked carefully against the source file (the input-file) and the appropriate correction to the software tool was made. This process was applied to certain files picked up beforehand (some of which are among the largest files in terms of the number of class features) in each system across the five Java systems, such as the *Parser* file in Barat. Once all the data of the aforementioned files were verified, the software tool was then run on the five Java systems

for the automated data collection. The same data was collected manually for each of the five Java systems.

3.3.7 Data analysis

Measurement is a fundamental concept in engineering. The conclusions of any empirical study are based on values measured on research objects. It is therefore crucial to assess the quality of the measurements and hence their conclusions. One of the most important properties of measurement is reliability, in the sense that the researchers can rely on the accuracy of the measuring agent (artifact or person). With respect to research conclusions inferential statistics are a tool that can help researchers give a quantitative estimate of the probable truth of the conclusions.

3.3.7.1 Statistical techniques

Reliability (Cronbach 1951) is the correlation of an item or scale with a supposed one which truly measures what it is supposed to. In view of the fact that there is no true scale, reliability can therefore be measured in different measurement methods. Popular reliability measurements are the inter-rater reliability coefficient of Cohen's kappa (Cohen 1960), and the internal consistency of Cronbach's alpha coefficient (Cronbach 1951).

In this Thesis, we have considered the aforementioned two reliability coefficients in order to investigate the level of reliability of the manually collected software metrics that are used as a partial basis of the research presented. In terms of inferential statistics (Field 2006) three tests were used, namely, one proportion test, two proportions test and the Kruskal-Wallis test.

3.3.7.1.1 Cohen's kappa

Inter-rater reliability is a measure of the level of agreement between different data-generating sources referred to as raters. The importance of an inter-rater reliability coefficient is that it can control the quality of the data collection method and help in generalising the research findings. Cohen (Cohen 1960) suggested the kappa statistic, considered as one of the best known inter-rater reliability coefficients (Viera and Garrett

2005). It provides researchers with a quantitative measure of the magnitude of agreement between independent observers (called judges in (Cohen 1960)) and is given by:

$$\kappa = \frac{p_O - p_C}{1 - p_C},$$

where p_o is the observed proportion of agreement, and p_c is the proportion of agreement expected by chance; k equalling one would imply a perfect agreement, and where k equals zero, a chance agreement.

In the context of this chapter, data is collected first manually and then automatically. Considering each type of data collection as a rater for these data, Cohen's kappa coefficient can then be applied to measure the level of agreement between these two raters, i.e., between data collected manually and then automatically. However, this test appears to be inapplicable for the manual and automatic data collection presented in the second part of this chapter. This is due to the fact that the kappa statistic could not be computed because it requires a symmetric 2-way table in which the values of the first variable (in our case the manual data collection) correspond to the values of the second variable (the automatic data collection) and that was not the case in most of the metrics considered. (In the terminology of Cohen (Cohen 1960, Cohen 1968) "categories" did not always match.)

3.3.7.1.2 Cronbach's alpha

Cronbach (Cronbach 1951) stated in his original work that:

> "Any research based on measurement must be concerned with the accuracy or dependability or, as we usually call it, reliability of measurement"

Reliability is estimated by examining the consistency with which different items express the same concept, rather than looking at consistency with which the same item is answered over time (DeVaus 2002).

Cronbach's alpha can be calculated either from the original item values or from standardised item values. The scores for standardised items have a mean of zero and a

variance of one. Raw-alpha data can be computed using the variance-covariance matrix computed from the item values, where the diagonal of the matrix contains the variance of each item and the rest of the matrix is composed of the covariances between all pairs of items (Bourque and Clark 1994, p. 73-74). The variance-covariance matrix of the standardised scores is the correlation matrix. The alpha formula (Cronbach 1951) using the variance-covariance matrix is as follows:

$$\alpha = \frac{n}{n-1}\frac{\sum_i \sum_j C_{ij}}{V_t}, \qquad (i, j = 1,2,\ldots n;\ i \neq j)$$

where n is the number of items, V_t is the variance of the total test and C_{ij} is the covariance of two items i and j. High values of alpha are more desirable, since a high alpha value is caused by a high V_t. Having a high variance value means that there is a widespread distribution of the scores. In our case it means that the errors of manual data collection are spread across all the system classes.

Cronbach's alpha is used in this chapter (see Section 3.4.6) to test the reliability of the manual collection of data in comparison with the automatic one.

3.3.7.1.3 One proportion test

The one proportion test is a hypothesis test to determine whether the proportion of trials that produce a certain event is equal to a target value. This procedure tests the null hypothesis that the population proportion (p) is equal to a hypothesised value (H_0: $p = p_0$). The alternative hypothesis can be one-tailed (($p < p_0$) or ($p > p_0$)), or two-tailed ($p \neq p_0$). In Chapter 4 (see Sections 4.5.4 and 4.5.5), this test is used to test the two one-tailed Hypotheses 3 and 4 and for each of which ($p > p_0$). The sample proportion *sample p* is calculated and the confidence interval lower bound and the exact p-value are computed for the tested proportion (Field 2006).

3.3.7.1.4 Two proportions test

The two proportions test is similar to the one proportion test. It is a hypothesis test for *two* population proportions to determine whether the difference between them is statistically

significant. This procedure uses the null hypothesis that the difference between two population proportions is equal to a hypothesised value (H_0: $p_1 - p_2 = P_0$), and tests it against an alternative hypothesis, which can be either one-tailed (($p_1 - p_2 < P_0$) or ($p_1 - p_2 > P_0$)) or two-tailed ($p_1 - p_2 \neq P_0$). For this test, the z value is calculated and two hypothesis tests are reported, which are the normal approximation test, and the Fisher's exact test. If the number of events in either of the tested samples is less than five then the normal approximation test may produce an inaccurate p-value, while the Fisher's exact test is accurate for all sample sizes (Collett 2003, Field 2006).

In Chapter 4 (see Sections 4.5.1 and 4.5.2), the two proportions test is used to determine whether the difference between the investigated proportions is statistically significant.

3.3.7.1.5 Kruskal-Wallis test

The purpose of the Kruskal-Wallis test is to investigate whether a set of K ($K \geq 2$) independent groups (samples) are drawn from populations with different median values (Field 2006). It is the extension of the non-parametric test of Mann-Whitney U test which involves more than two independent samples. Therefore, if we applied Kruskal-Wallis to two independent samples, the results will be equivalent to those obtained with the Mann-Whitney U test. If we have K independent samples of sizes $n_1, n_2, \ldots, n_k$ we combine all the samples into one large sample, sort the result from smallest to largest and assign ranks (assigning the average rank to any observation in a group of tied observations). The rank order statistic H for the k-sample problem is:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{K} \frac{R_i^{\,2}}{n_i} - 3(N+1),$$

where $i = 1,2,\ldots,K$, n_i is the number in the ith sample, N is the total $\sum n_i$ and R_i is the sum of ranks for the ith sample. This test has a degree of freedon (df) which is one less than the number of the tested samples (K -1). If the result of the Kruskal-Wallis test is significant ($p < 0.05$), it indicates that there is a significant difference between at least two sample medians of the K medians. Consequently, it can be concluded that there is a high propability that two of the samples, at least, represent populations with different medians

(Sheskin 2004). In Chapter 6, Kruskal-Wallis is used to test the significance of difference in the five Java software systems (see Section 6.6.2) in terms of the number of comment lines that surrounded, or are embedded within, constructors.

We note that the SPSS (V.11.5) and the Minitab (V.15) statistical packages are used to produce the required statistics in this Thesis.

3.3.7.2 Drawing conclusions and the generalisation issue

The adopted methodology in generalising the results of the research presented in this Thesis is based on two themes. Firstly, the testbed systems were chosen from different application domains with a wide range of sizes for the purpose of gaining a greater understanding of the differences between such systems from our research problem perspectives. Secondly, using the inferential statistical techniques (such as the one proportion test and the Kruskal Wallis test) is of great help for generalising the research results, since such statistical techniques use inductive reasoning. Furthermore, the reliability method of Cronbach's alpha is used to support the generalisation issue from the perspective of manual data collection.

Since our empirical studies depend on data collected sometimes automatically and on other occasions manually, we found it necessary and useful to investigate to some extent the difference between automatic and manual data collection. (A full investigation of this issue is outside the scope of the Thesis.)

3.4 Manual and Automatic Collection of Data

Collecting metrics and then analysing the resulting data can provide a variety of insights into trends, features and habits used in the construction of software artifacts (Harrison et al. 1998a, Harrison et al. 1998c, MacDonell and Shepperd 2003). The quality of OO software (and systems generally) from a maintenance and hence evolutionary point of view is largely

down to ensuring that their features conform to sound and accepted practice (Kitchenham and Pfleeger 1996).

Collection from OO software artifacts can be done either manually, or, using appropriate software, automatically. While we accept that there are various advantages to collecting such data automatically, i.e., speed and convenience, there are numerous occasions when manual data collection is necessary and unavoidable. This is particularly true when, firstly, on-line documentation is no longer available and hard copy is the only available source; secondly, when it is necessary to check and verify the results of an automatic collection; thirdly, when the artifacts (in machine readable format) may be graphical in nature and thus difficult to machine-read. We also need to consider an important and emerging OO issue, namely, the identification from code of design patterns (Gamma et al. 1995), a task which can only be done effectively through manual observation and collection of the relevant classes (Bieman et al. 2003). In this case, software tools to facilitate this task do not currently exist.

Anecdotally, manual data collection has been viewed as both a highly error-prone and time-consuming process; we do not dispute this claim. However, very little empirical evidence exists to substantiate the *extent* of the error-proneness associated with manual data collection and hence the lack of quality and confidence in the data produced manually. In this empirical study, the same data from five Java systems was collected both automatically and manually and the differences between the two resulting datasets were analysed. A number of metrics, derived from the source code, were used as the basis for our data analysis.

A key indicator of the manual data collection was found to be due to developer coding style where class features were un-ordered and arranged in a haphazard way; large classes in some of the systems also proved to be a cause for erroneous manual data collection. The least error-prone system was an OO *framework* and the most error-prone was the largest of the five Java systems, Swing, which, in other empirical studies (see Section 6.4), has consistently been shown to violate accepted OO practice. From the results hereafter, we thus hypothesise that relatively small, well-arranged classes are of benefit in terms of

software maintenance and reengineering with further implications for program comprehension. We also conclude, on the empirical evidence available, that the quality of data we extracted manually from the five Java systems is a direct reflection of the quality of the systems themselves.

3.4.1 Motivation and related work

In the following sections of this chapter, we present details of an empirical study regarding the manual collection of data vis-à-vis its automatic counterpart; various qualitative and quantitative analyses have been used in the past (Schneidwind 1992, Basili et al. 1996, Briand et al. 1997a, Counsell et al. 2000, El Emam et al. 2001, Darcy et al. 2005) to support studies such as the one presented in the sequel.

The key motivation for this study is to establish just how error-prone manual data collection can be when compared with its automatic counterpart. Herein, we explore the possibility that manual data collection is not significantly more error-prone than an automatic collection of the same data; we also explore whether large systems are more susceptible to manual data collection errors than relatively smaller systems and, finally, whether errors are generally due to under-counting or over-counting by the data collector.

The choice of metrics for our study was made on the basis that changes to classes relate to the attributes and methods of those classes. Although it is true to say that systems do grow through addition of lines of code *per se*, significant numbers of attributes and methods generally get added throughout the lifetime of an OO system. The metrics collected as part of this study thus covered the different types of attribute and method declaration, whether private, protected or public. Manual collection of these types of metrics is common in empirical studies where often the on-line versions of the software are not available. Software metrics are an integral part of any empirical study (Fenton and Pfleeger 2002) and guidelines for data analysis techniques and conducting empirical studies properly have been proposed by Kitchenham et al. (Kitchenham et al. 2001, Kitchenham et al. 2002).

In (Counsell et al. 2002) the question of encapsulation and inheritance in five different C++ systems was addressed. Metrics were automatically collected for three of the systems

analysed therein. However, the remaining two systems were not available electronically and, as a result, manual collection from paper sources was unavoidable. Interestingly, common trends were found across all five systems, suggesting that manual data collection does have certain merits.

This empirical study has, in addition, implications for refactoring in some cases (Fowler 2000), since visual understanding and assessment of code is the first stage of any refactoring. In (Najjar et al. 2003), an empirical investigation of an OO refactoring was described where constructors were replaced with methods and those methods were then renamed to make them more meaningful. Part of the process of replacing those constructors was to count the number of identical lines between *n* constructors and also to rename those constructors to non-constructor methods; both activities, particularly the latter were almost impossible to complete automatically and so manual data collection was essential in that case. Finally, Fenton and Pfleeger (Fenton and Pfleeger 2002) describe manual recording as a subject to bias, whether deliberate or unconscious, but they do admit that sometimes there is no alternative to manual data collection.

3.4.2 Empirical investigation

Our empirical study consisted of five Java systems described in Section 3.3.3.2. A *key assumption* that we make in our study of the five systems is that the automatic data collection represents the *correct* values of the metrics under consideration. To support this assumption, results from the automatic collection were meticulously verified by visual inspection. Java software was written (see Appendix B) to collect the data automatically and this software run after the manual data collection had finished. Both automatic and manual collections were carried out by the author. The said assumption forms the benchmark against which we compare the manually collected data.

In the sequel, the terms *class features* or simply *features* will be used interchangeably; also, the size of a system is taken to be the number of its classes.

3.4.3 Data collected

In total, fifteen class metrics were collected, see Section 3.3.5. The metrics represent three Java categories: metrics related to private features, metrics related to protected features and metrics related to public features. Herein, we consider every attribute and return type of a method as a primitive attribute or primitive method, respectively, if their types are taken from one of the following:

int, long, float, double, short, char, byte, boolean, and *void.*

A non-primitive attribute is one which is defined as a class; again this is normally another class, but can equally be the class in which the attribute is defined. We define a non-primitive method as one whose return type is that of a class (which is normally a different class, but can equally be the same class in which the method is defined). We chose to categorise attributes and methods to this extent to reflect the fact that in most OO systems, attributes tend to be declared private and methods public.

We chose to use attributes and methods as a basis for our study since they represent the quintessential elements of an OO class. We note that in collecting data manually, it is possible for a metric to be under-counted (we will, henceforth, call this a *negative error*) and also to be over-counted (we will, henceforth, call this a *positive error)*. We thus define a further four system metrics to aid our overall analysis (taking account of these two possibilities):

16. The **negative error size (NES)** metric: the sum of negative errors for a metric.
17. The **negative error frequency (NEF)** metric: the total number of times that negative errors occurred for that particular metric.
18. The **positive error size (PES)** metric: the sum of positive errors for a metric.
19. The **positive error frequency (PEF)** metric: the total number of times that positive errors occurred for that particular metric.

3.4.3.1 Example using priPA

To inform an understanding of the four metrics described earlier, consider the example of a system, say *S*, containing only three classes, *X*, *Y* and *Z*, with the metric priPA computed automatically and equal to 9, 11 and 15, respectively.

Let us assume that the data collector counts 9, 8 and 13 priPA, respectively. The NES would thus equate to -5 representing an under-count of 3 for class *Y* and an under-count of 2 for class *Z*. The NEF is thus 2 (i.e., a negative error was made on two occasions). Correspondingly, PES and PEF can be similarly evaluated. For example, if the data collector counts 10, 13 and 16 for *X*, *Y* and *Z*, respectively, then PES = 4 and PEF = 3.

In the sequel, the data collected for these four metrics is presented compactly in summary format in terms of NES, NEF, PES and PEF.

3.4.4 Three hypotheses

Herein, we investigated three hypotheses. We have expressed each in terms of a null and alternative hypothesis. We denote a null hypothesis by H_{0x} to distinguish it from the alternative hypothesis denoted H_{Ax}. Enumerated, the null and alternative hypotheses are:

- Hypothesis one

  H_{01}: There is no quantitative difference between the errors made in the manual collection of data from large systems vis-à-vis relatively smaller systems.

  H_{A1}: Manual data collection from large systems is more error-prone than manual data collection from relatively smaller systems.

This is based on the belief that larger systems have undergone more maintenance than relatively smaller systems; consequently they will exhibit a less organised layout of classes.

- Hypothesis two

  H_{02}: Errors made in the manual collection of data are consistent in terms of a) number of under-counts and b) number of over-counts.

H_{A2}: Manual data collection will always tend to under-count rather than over-count class features. In other words, during manual data collection, class features will be under-counted by the data collector rather than over-counted.

- Hypothesis three

  H_{03}: There is equal likelihood of the data collector making manual data collection errors in the collection of either private, protected or public class features within each of the five systems.

  H_{A3}: There are large differences in the errors associated with the manual collection of either private, protected or public class features within each of the five systems.

  This hypothesis is based on the belief that, syntactically, the layout and style of declarations of the three access types within each system differs widely. Consequently, the data collector is likely to make errors in favour of certain access types.

Henceforward, we refer to the hypotheses as null hypothesis H_{0x} or alternative hypothesis H_{Ax}; we also refer to hypothesis one, two and three when we mean both the null and alternative hypotheses. We adhere to this convention throughout the Thesis.

3.4.5 Data analysis

After collection of the relevant data, an analysis of the differences between manual and automatic collection was conducted. For example, Table 3.2 shows the size of the errors from the manual collection for each of the metrics considered.

The minimum (Min) column represents the minimum value of the number of negative errors made in the manual collection for that particular metric. For example, -5 signifies the largest single negative error for the priPA metric. Similarly, the maximum (Max) column represents the maximum value of the number of positive errors made for that particular metric. For example, 4 signifies the largest single positive error made for the priNPA metric. We note that some rows of Table 3.2 are omitted for metrics where the differences between manual and automatic are zero (or the corresponding metrics values are zero for both manual and automatic data collection).

The values for priPA and priNPA show relatively large negative and positive errors, respectively. The reason for these errors is primarily due to the way that attributes were defined in GraphDraw (as opposed to the way one would expect them to be defined in a normal system). This took the form:

```
String x = new String("foo"), y = new String("foo1")....;
```

The usual coding convention is to arrange attributes on separate lines; it seems that the syntax adopted for this system may have been the cause of errors in the manual data collection.

GraphDraw Metric Differences						
	NES	NEF	PES	PEF	Min	Max
priPA	-5	1	1	1	-5	1
priNPA	-1	1	4	1	-1	4
priC	-1	1	0	0	-1	0
priPM	-1	1	0	0	-1	0
priNPM	0	0	1	1	0	1
proPA	-1	1	0	0	-1	0
pubNPM	-2	1	0	0	-2	0

Table 3.2: Differences between automatic and manual metrics for GraphDraw

An interesting feature from Table 3.2 is the relative lack of errors in protected and public members. Visual inspection of the classes revealed the protected and public declarations for GraphDraw to be well ordered and easily distinguishable from the other class features; they were thus collected relatively free from error.

Table 3.3 shows the number of the errors for the BSF (framework) system. The most error-prone metrics relate to public features, although the general trend in BSF is for low errors across all metrics. Interestingly, the rows of protected metrics are absent from Table 3.3. A number of explanations may clarify why this is the case.

In Chapter 4 (cf. Counsell et al. 2002) the ET++ framework system was shown to be the most conformant with sound OO encapsulation principles. We would thus generally expect classes outside any inheritance hierarchy to contain zero protected features and classes inside an inheritance hierarchy to contain large numbers of protected features.

En passant we note that in Chapter 4 five C++ systems are investigated; large numbers of protected features are found in classes outside any inheritance hierarchy for four of the five systems. The only exception to that trend is the ET++ framework (Weinand et al. 1988). Chapter 4 (see Section 4.5.2) and the current findings for the BSF framework system, therefore, support previous findings about frameworks. Tentatively, we could conclude that the better written system has caused relatively fewer manual errors.

BSF Metric Differences						
	NES	NEF	PES	PEF	Min	Max
priNPA	-1	1	0	0	-1	0
priPM	-1	1	0	0	-1	0
pubNPA	-1	1	1	1	-1	1
pubPM	-3	2	1	1	-2	1
pubNPM	0	0	2	2	0	1

Table 3.3: Differences between automatic and manual metrics for BSF

Noticeable from Table 3.4 is the number of negative errors, particularly for the pubNPM metric. The NES value for this metric was -10 with frequency 9, suggesting that on average roughly one method was overlooked on each occasion. One reason to explain the difficulty of collecting data manually from Barat is that it may have evolved to a far greater extent than the other systems with more maintenance having been applied to it than BSF and GraphDraw. Private, protected and public metrics all show errors in their manual collection. A number of reasons may explain why this is so.

Barat Metric Differences						
	NES	NEF	PES	PEF	Min	Max
priPA	-1	1	2	2	-1	1
priNPA	-4	4	0	0	-1	0
priNPM	0	0	1	1	0	1
proPA	0	0	1	1	0	1
proNPA	-1	1	1	1	-1	1
proC	-1	1	0	0	-1	0
proPM	0	0	3	3	0	1
proNPM	-3	3	0	0	-1	0
pubPA	-1	1	0	0	-1	0
pubNPA	-1	1	1	1	-1	1
pubC	-1	1	1	1	-1	1
pubPM	-5	5	2	2	-1	1
pubNPM	-10	9	0	0	-2	0

Table 3.4: Differences between automatic and manual metrics for Barat

Firstly, collection of attributes and methods in this system was fraught with difficulty because of a tendency by the developers to interleave private, protected and public declarations, thus increasing the possibility of overlooking such declarations in the manual collection process. As an example, a series of public method declarations would be followed by a series of private method declarations, followed by more public method declarations, etc. As well as making the task of the data collector difficult, this would hamper the task of any maintainer should changes be needed to these class features. We would claim that this is a sign of poor maintenance (and possibly degradation during evolution).

Secondly, there was a wide variation in the *style* of layout of Barat's classes, which made those classes even more difficult to follow and collect data from. This may be due to different developers enhancing those classes (or even originally developing those classes in that way). The lack of coding standards for this system may be one contributing factor in the error-proneness of the manual data collection.

Thirdly, Barat also contained a high proportion of very large classes (one such class covered eighty-nine A4 pages, portrait format).

These three factors combined to make the manual data collection relatively error-prone for Barat. As a result, the quality of the data collected suffered.

Further evidence of the poor structure of the Barat system can be found in Chapter 6 (see Section 6.4) (cf. Najjar et al. 2003), where Barat was found to have the highest frequency of large numbers of constructors amongst its classes (when compared with the same four other Java systems). In contrast, it is also noted, in the same chapter, that the BSF system has the lowest value for the same frequency. We suggest that a class with a high frequency of large numbers of constructors is evidence of poor programming practice, since from a maintainability point of view, the class becomes difficult to understand, causes developers to add yet more constructors and hence is a prime candidate for refactoring (Fowler 2000, Kerievsky 2004). In this respect, the quality of the Barat system seems to be relatively poor vis-à-vis BSF and GraphDraw.

For the private and public features of Libjava, there is some evidence of discrepancies between the automatically and manually collected data. Table 3.5 shows a relatively high NES value for pubPM. We would expect the Libjava system to contain a high proportion of public methods as part of the library classes' definition; the designers of the library classes would also want the developer to have as much freedom of access to, and modifiability of, the methods in the library classes as possible.

Libjava Metric Differences						
	NES	NEF	PES	PEF	Min	Max
priC	-1	1	0	0	-1	0
priPM	0	0	1	1	0	1
pubC	-1	1	1	1	-1	1
pubPM	-7	5	1	1	-2	1
pubNPM	-1	1	1	1	-1	1

Table 3.5: Differences between automatic and manual metrics for Libjava

In Table 3.6 for the Swing system, there appear to be widespread errors in the manual data collection. One explanation for this result may be that for Swing (as with the Barat system) the classes have had larger and more frequent changes applied to them. In turn, this has caused the classes to deteriorate from their original form. Such a deterioration may have contributed to errors in the process of manual data collection. The values for both the sum of errors and maximum errors point to public features being the most erroneous. It is interesting that the minimum and maximum errors for the pubPM metric are -7 and 7, respectively. This suggests that frequent errors were made through both under-counting and over-counting.

Swing Metric Differences						
	NES	NEF	PES	PEF	Min	Max
priPA	-12	10	7	4	-3	3
priNPA	-17	14	7	7	-2	1
priPM	-3	2	3	2	-2	2
priNPM	-4	3	7	6	-2	2
proPA	-6	5	2	2	-2	1
proNPA	-12	8	0	0	-4	0
proPM	-12	9	4	3	-3	2
proNPM	-5	5	3	3	-1	1
pubNPA	-8	5	1	1	-3	1
pubC	-4	4	1	1	-1	1
pubPM	-25	18	16	8	-7	7
pubNPM	-18	14	12	8	-3	3

Table 3.6: Differences between automatic and manual metrics for Swing

3.4.6 Hypotheses re-visited

For the reader's sake, we re-state each of the three null and alternative hypotheses.

3.4.6.1 Hypothesis one re-visited

- H_{01}: There is no quantitative difference between the errors made in the manual collection of data from large systems vis-à-vis relatively smaller systems.
- H_{A1}: Manual data collection from large systems is more error-prone than manual data collection from relatively smaller systems.

Table 3.7 shows Cronbach's alpha coefficient values across the five Java systems. A *na* entry in the table reflects the fact that only zero values existed for that particular metric within the system (and hence no alpha value could be computed). The values of alpha are greater than 0.9 (except for one case) indicating that the two methods of data collection are equivalent (Field 2006). Using the evidence from Table 3.7, the least error-prone systems on balance appear marginally to be Libjava and BSF; each of these systems has 9 perfect alpha values (the most error-prone is Swing, which has 12 imperfect alpha values). Using the number of classes as the size of a system, Libjava is the third largest of the five systems. When Barat and Swing (the two largest and most error-prone of the five systems) are benchmarked against Libjava, the data does support H_{A1}, whilst when GraphDraw and BSF are benchmarked against Libjava, the data does not support H_{A1}.

Cronbach's Alpha Coefficient					
	GraphDraw	BSF	Barat	Libjava	Swing
priPA	0.9976	1.0000	0.9983	1.0000	0.9973
priNPA	0.9972	0.9997	0.9990	1.0000	0.9988
priC	0.7968	1.0000	na	0.9700	1.0000
priPM	0.9998	0.9953	1.0000	0.9979	0.9990
priNPM	0.9953	1.0000	0.9982	1.0000	0.9922
proPA	0.9987	1.0000	0.9971	1.0000	0.9979
proNPA	1.0000	1.0000	0.9988	na	0.9987
proC	na	na	na	1.0000	1.0000
proPM	1.0000	1.0000	0.9968	1.0000	0.9996
proNPM	na	1.0000	0.9482	1.0000	0.9992
pubPA	1.0000	1.0000	0.9930	1.0000	1.0000
pubNPA	1.0000	0.9791	0.9957	1.0000	0.9998
pubC	1.0000	1.0000	0.9991	0.9975	0.9993
pubPM	1.0000	0.9988	1.0000	0.9995	0.9995
pubNPM	0.9980	0.9992	0.9998	0.9998	0.9994

Table 3.7: Cronbach's alpha coefficients for manual and automatic data collection metrics of the five Java systems

However, in order to test the significance difference between the large software systems group (Swing and Barat) and the small systems group (GraphDraw, Libjava and BSF), the non-parametric Mann-Whitney test (Field 2006) is used to compare the median error rates from the aforesaid two groups, for each metric collected from the five Java systems.
Table 3.7a shows the Mann-Whitney statistic U, z value and the significance for each metric considered. From Table 3.7a we can conclude that the large software systems group under consideration does not significantly vary (> 0.05 in most of the cases) from the small systems group in terms of median error rates.

Therefore, there is not enough evidence to reject the null hypothesis (H_{01}). We conclude that there is no quantitative difference between the errors made in the manual collection of data from large systems vis-à-vis relatively smaller systems.

	Mann-Whitney U	Z value	Asymptotic significance	Exact significance
priPA	169958.00	-0.40	0.34	0.49
priNPA	170155.00	-0.20	0.42	0.40
priC	168810.00	-4.01	0.00	0.01
priPM	169639.00	-1.07	0.14	0.11
priNPM	170051.50	-0.43	0.33	0.49
proPA	169845.50	-0.71	0.24	0.39
proNPA	169641.00	-0.90	0.19	0.35
proC	170362.00	-0.35	0.36	0.89
proPM	170156.00	-0.27	0.39	0.43
proNPM	169950.00	-0.53	0.30	0.40
pubPA	170362.00	-0.35	0.36	0.89
pubNPA	170054.00	-0.47	0.33	0.53
pubC	170157.50	-0.35	0.36	0.60
pubPM	167551.00	-1.54	0.06	0.04
pubNPM	168130.50	-1.35	0.09	0.13

Table 3.7a: The Mann-Whitney test statistics for comparing the mean error rates from the large and small software systems groups

3.4.6.2 Hypothesis two re-visited

- H_{02}: Errors made in the manual collection of data are consistent in terms of a) number of under-counts and b) number of over-counts.
- H_{A2}: Manual data collection will always tend to under-count rather than over-count class features. In other words, during manual data collection, class features will be under-counted by the data collector rather than over-counted.

Table 3.8 shows the total of the negative and positive error values for the five systems. Clearly, the Swing system is the most error-prone, and BSF the least error-prone.

Summary Table						
System	NES	NEF	PES	PEF	Min	Max
GraphDraw	-11	6	6	3	-5	4
BSF	-6	5	4	4	-2	1
Barat	-28	27	12	12	-2	1
Libjava	-10	8	4	4	-2	1
Swing	-126	97	63	45	-7	7

Table 3.8: Total values for errors made for all five Java systems

From the data in Table 3.8 we can see that the sum of negative errors made over the five systems outstrips that of the positive errors. The total frequency of the negative errors is also higher than that of the positive errors in every case. We thus conclude that, given the evidence in Table 3.8, H_{A2} can be supported by the data.

3.4.6.3 Hypothesis three re-visited

- H_{03}: There is equal likelihood of the data collector making manual data collection errors in the collection of either private, protected or public class features within each of the five systems.
- H_{A3}: There are large differences in the errors associated with the manual collection of either private, protected or public class features within each of the five systems.

Remarkable from Table 3.7 are the perfect alpha vales for the complete set of protected features for both BSF and Libjava. The alpha values for public declarations for Graphdraw, on the other hand, are all perfect except pubNPM; this system has the highest number of perfect alpha values for public declarations vis-à-vis private or protected (or both). It can be seen from this table that the number of perfect alpha varies widely between systems.

Both the BSF and Libjava systems have also the highest number of perfect alpha values for the private declarations vis-à-vis the public ones. All three access types show wide variation within each system. There is thus no obvious pattern to the empirical results. Based on the evidence from Table 3.7 we support H_{A3}.

3.4.7 Cost versus accuracy

Measuring the accuracy of the manual data collection process provides a valuable insight into the types of error a data collector is likely to make. Careful consideration, however, needs to be given to the cost of collecting that data against the accuracy that the said process offers.

The data collector in this empirical study spent the equivalent of three months full-time on collecting the data from the five systems. Working five days a week, seven hours a day for twelve weeks equates to a total of 420 hours. If we say, for argument's sake, that the cost of collection in pounds/dollars is x per hour, then the total cost of collection is $420x$ pounds/dollars.

If we then make the very broad assumption that each class takes the same amount of time to inspect and collect the fifteen metrics from, the five systems would have absorbed costs in the ratios 52:1861 (3%), 65:1861 (3%), 407:1861 (22%), 89:1861 (5%) and 1248:1861 (67%), for Graphdraw, BSF, Barat, Libjava and Swing, respectively; 1861 is the sum of all classes across the five systems. If we then allocate the $420x$ costs according to these ratios, we get approximately: $13x$, $13x$, $92x$, $21x$ and $281x$ for the same five systems.

Next, we need to consider the sum of errors made in each system, from which we can then assess the average cost in x's for each error made. The sum of errors made for the five systems (in the same order as previously stated) were: 17, 10, 40, 14 and 189. This means that, on average, the cost of an error in Graphdraw was $13x/17$ ($0.76x$), in BSF $13x/10$ ($1.30x$), in Barat $92x/40$ ($2.30x$), in Libjava $21x/14$ ($1.50x$) and in Swing $281x/189$ ($1.48x$).

From this analysis we can then conclude that the Graphdraw system provides most value for time expended (only $0.76x$ was expended by the data collector for each error made in this system). The least value was for the Barat system, where it cost $2.30x$ for each error made by the data collector. In other words, given the choice between the five systems, the time of the data collector was best spent collecting data manually from the Graphdraw system, and least effective when collecting data from the Barat system. Of course, we have

also made the assumption in our analysis that each error made by the data collector incurs an equivalent amount of cost.

As an epilogue, a wide range of errors were made by the data collector (whether through under-counting or over-counting), so in reality the cost allocation may be different. One interesting aspect of future work would be to associate a level of complexity to each error made and then carry out a form of weighted analysis.

3.4.8 Discussion

In a study of this type, we have to carefully consider the caveats to the validity of the study and the conclusions thereof. The first caveat comes from the assumption that values from the automatic data collection are the correct values. While we do see the possibility that the automatic collection can produce incorrect values, significant effort was invested in making sure (see Section 3.3.6) that the correct values had been obtained, and that the software employed was as fault-free as possible.

A further caveat comes from the belief that it is the size of the Swing system which made it the most cumbersome of the five systems to collect data from. This caveat could be criticised, since the error-proneness may be due merely to fatigue on the part of the data collector. While this may be true, the collection was not helped by the poor layout and inconsistent ordering of the private, protected and public class features as well as other poor coding practices. We therefore suggest that it would have been the most error-prone system of the five systems *even* allowing for significant fatigue effects. Also, from a time point of view, the systems were analysed on a consistent and regular basis over a six-month period and not according to their size.

Finally, it was surprising to discover how few errors were made (and, indeed, the number of those errors) as part of the manual collection. Only on one or two occasions were the metrics values very different from those of the automatic collection. This may be due to the simplicity of the metrics chosen; more complex metrics may prove more error-prone. An aspect of future work will focus on this issue.

While these empirical results go some way to dispelling the myth about the error-proneness of manual data collection, more studies need to be undertaken before any concrete conclusions can be drawn. It is a worrying scenario that poorly written and/or maintained systems seem to pose the most difficult problems for manual data collection. We would speculate that for maintainers of the code, similar difficulties do arise.

3.5 Summary

Two main themes are discussed in this chapter. The first theme addresses the rationale for the decisions made concerning research methodology. The second theme of this chapter pertains to the investigation of the extent to which manual data collection compares with its automatic counterpart when gathering the same data for the five Java systems.

In this chapter, we have provided a discussion on the research methodology. The research presented in this Thesis is quantitative and uses software artifacts. Ten C++ and Java software systems were selected purposively according to a number of criteria. Justification for how the selection of the research methods specifically supports achieving our objectives was presented and how this impacts generalisation. The statistical techniques that are used throughout the Thesis are also presented.

The second part of this chapter is concerned with investigating the extent to which manual data collection compares with its automatic counterpart. We found that poor programmer habits, disorganised code and large classes seemed to be the real reasons why manual data collection was error-prone and departs from its automatic counterpart. It was also found that a data collector would tend to under-count rather than over-count class features during data collection (Hypothesis two). This empirical study provides an insight into some of the typical problems associated with collecting data manually and highlights possible coding pitfalls, which may cause problems from a reengineering and maintenance perspective.

We conclude from this empirical investigation and that, from a quality perspective, a poorly written system will contribute to errors in any manual data collection. In other words, the

quality of such data reflects how well the system itself was coded. From a practical perspective, it would seem important for developers to always adhere to recognised (or in-house) coding standards. Failure to do so may cause problems associated with maintaining the software and, in turn, compromise its quality as it evolves.

Since the core of this Thesis is to investigate how encapsulation is interpreted, improved upon and put into practice, the following chapter will empirically investigate the trends of encapsulation in the ten C++ and Java systems referred to earlier in this chapter (see Section 3.3.3.2).

CHAPTER 4 Encapsulation Trends in C++ and Java Software Systems

4.1 Introduction

In the previous chapter we described our research strategy and how we made our decisions about the main issue of the empirical research. In Chapter 3, we also empirically investigated the extent to which manual data collection can differ from its automatic counterpart.

In this chapter, we present an empirical study in which encapsulation and inheritance trends from the ten C++ and Java systems (introduced and described in Chapter 3) were examined from collected data. Encapsulation and inheritance are two of the cornerstones of the OO paradigm. In fact, we could view their relationship as symbiotic, since proper use of inheritance requires the developer to consider protected declarations as standard design practice. We would also expect developers to consider the use of private methods in inheriting classes in order to prevent subclasses accessing certain behaviour of superclasses (parent classes).

In Section 4.2 we describe how encapsulation and inheritance are related to each other theoretically. In Section 4.3 we present the motivation for the undertaken work and in Section 4.4 we present the details of the empirical evaluation, including a statement of four hypotheses. In Section 4.5 we provide a description of the data analysis undertaken. A discussion of the issues raised in this chapter is given in Section 4.6 including possible caveats to the validity of the study and the conclusions thereof, and we provide a summary for this chapter in Section 4.7.

We note that part of the research in this chapter, dealing with the C++ systems, was published in (Counsell et al. 2002).

4.2 Encapsulation and Inheritance

Encapsulation and inheritance are two of the cornerstones of the OO paradigm. *Inheritance* lies at the heart of the OO paradigm. It is the mechanism that allows a class *b* to inherit all of the attributes and methods provided by a class *a*; all the data structures and methods or operations designed and implemented for *a* are available for *b*. In other words, inheritance is the mechanism that creates a new class from an existing one. Access to the inherited data and methods is regulated by access specifiers provided by encapsulation. *Encapsulation* is a technique that encloses data and operations performed on the data within one class and provides the access control to the remaining classes of the system by specifying which class members should be private, which protected and which public.

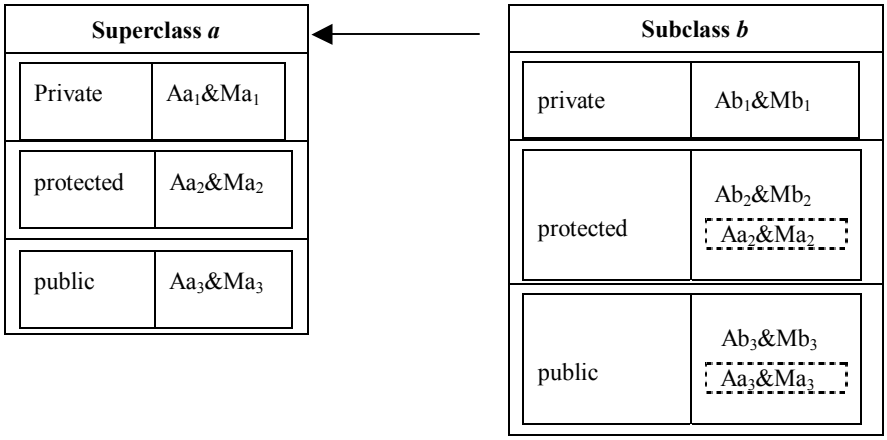

Figure 4.1: Relationship between encapsulation and inheritance

Figure 4.1 describes the relationship between encapsulation and inheritance for class '*a*' and class '*b*'. All the protected (also the public) members which belong to '*a*' are directly used by its subclass '*b*'. So class '*b*' has the advantage of using the protected and public members (Aa_2&Ma_2 and Aa_3&Ma_3) of its superclass '*a*'. At the same time, it has new members (Ab_1&Mb_1, Ab_2&Mb_2 and Ab_3&Mb_3), attributes and methods, which are required to complete its definition. In the C++ and Java languages, class members can be declared as private, protected or public.

In C++, public members of a class can be accessed by any function. We note that in C++ there is the concept of a *global* function and data which do not belong to any class in the

system. Throughout this Thesis the terms 'function' and 'method' are used interchangeably. A private member is only accessible by methods that are members of the class under consideration and by classes and methods explicitly granted access permission by the *friends* facility (Stroustrup 1991). A protected member can be accessed by members of classes that inherit from the class in addition to the class itself (and any friend). So, friends, whether they are classes or methods, are allowed to access both private and protected members. Also, in C++, a class member is private by default.

In Java, the private members of a class can only be accessed by their class members. Protected members are accessible from anywhere within the *package* in which they were declared, and by any classes which inherit from the class containing the said protected members (Eckel 2000). As in C++, public members are accessible to all remaining classes in the system. Java has a default access, called *package access*, which facilitates access to all the classes defined within the same *package*.

In the following section, the motivation for, and related issues of, our empirical study are described.

4.3 Motivation and Related Issues

A large body of work has been carried out to investigate the influence of inheritance on the way we write OO systems and, to a much lesser extent, that of encapsulation. Even fewer studies have investigated the relationship between the two. Encapsulation and inheritance play a fundamental role in the OO paradigm**.** Inheritance is a mechanism that creates a new class from an existing class, providing attributes and methods in the created class without affecting existing classes. Encapsulation is a way of controlling the visibility of attributes and methods of a class from inside and outside. Practically, encapsulation controls the visibility of attributes and methods by providing one of the access specifiers (private, protected, or public). In well-designed OO software, any changes to a method's implementation in an object should have no impact on other objects; however, sometimes achieving no impact on other objects is difficult for many reasons. We believe that, in

practice, encapsulation and inheritance are not properly used in OO in a mutually supportive way**.** Several studies have investigated the benefits and the use of encapsulation and inheritance, and the relationships between them in different OO languages (Snyder 1986, Post 2001, Skoglund 2003, Schärli et al. 2004).

In this chapter, we present, as a first step of our empirical research, a study in which encapsulation and inheritance trends from the C++ and Java systems, described in Chapter 3, were examined from relevant collected data. Trends in the use of class features declared as private, protected and public for both inheriting and non-inheriting classes were investigated.

4.4 Empirical Investigation

We investigated the trends of encapsulation through four hypotheses; two were investigated for the C++ software systems and the other two for the Java systems. We next describe the details of these hypotheses.

4.4.1 The hypotheses

In the description of the two hypotheses for the C++ systems, we have expressed each hypothesis in such a way that its opposite is what we would expect to find according to proper OO design principles. The hypotheses support our intuition about what really happens in OO systems. We will therefore seek to support the hypotheses through standard statistical analysis. Each of these four hypotheses is expressed in terms of a null and an alternative hypothesis. The first two hypotheses are for the C++ software systems:

- Hypothesis one:

  H_{01}: There is no quantitative difference between the proportion of privately declared methods (as opposed to protected and public) of classes which engage in inheritance from those which do not.

H_{A1}: Classes which engage in inheritance have a lower proportion of privately declared methods (as opposed to protected and public) than classes which do not engage in inheritance.

This hypothesis is based on the belief that the software developer should encapsulate (private) attributes and methods which are not supposed to be inherited by subclasses in the inheritance hierarchy. However, we reject the (perhaps) accepted notion (Stroustrup 1991) that classes which engage in inheritance should have a higher proportion of privately declared attributes and methods than classes which do not engage in inheritance.

Hypotheses two, three and four are expressed in terms of two sub-hypotheses.

- Hypothesis two:

  H_{02i}: There is no quantitative difference between the proportion of protected attributes (as opposed to private and public) of classes which engage in inheritance from those which do not engage in inheritance.

  H_{A2i}: Classes which engage in inheritance have a lower proportion of protected attributes (as opposed to private and public) than classes which do not engage in inheritance.

  H_{02ii}: There is no quantitative difference between the proportion of protected methods (as opposed to private and public) of classes which engage in inheritance from those which do not engage in inheritance.

  H_{A2ii}: Classes which engage in inheritance have a lower proportion of protected methods (as opposed to private and public) than classes which do not engage in inheritance.

This hypothesis is based on the belief that, in theory, as part of using inheritance, the software developer should encapsulate attributes and methods (whilst permitting inheritance of data and behaviour) through protected declarations. However, we feel that this is not generally the case in C++ systems. We therefore reject the (perhaps) accepted notion (Stroustrup 1991) that classes which engage in inheritance should have a higher

proportion of protected attributes and methods than classes which do not engage in inheritance.

For the Java systems, we propose the following two hypotheses:

- Hypothesis three:

  H_{03i}: There is no quantitative difference between the proportion of private attributes and the proportion of public attributes in Java software systems.

  H_{A3i}: The proportion of private attributes is greater than the proportion of public attributes in Java software systems.

  H_{03ii}: There is no quantitative difference between the proportion of protected attributes and the proportion of public attributes in Java software systems.

  H_{A3ii}: The proportion of protected attributes is greater than the proportion of public attributes in Java software systems.

This hypothesis is based on the theoretical premise that every class in Java is a subclass of at least the *Object* class. So we would expect to see that the smallest number of attributes declared is public, and the largest one is private. We would also expect to see more declaration of protected attributes than public ones; if a class is part of an inheritance hierarchy, attributes should be declared as protected to facilitate access to inherited classes and prevent access to them from outside the inheritance hierarchy.

- Hypothesis four:

  H_{04i}**:** Classes at the top of an inheritance hierarchy or at the bottom have equal proportion of protected attributes.

  H_{A4i}: Classes at the top of an inheritance hierarchy have a greater proportion of protected attributes than classes at the bottom.

  H_{04ii}**:** Classes at the top of an inheritance hierarchy or at the bottom have an equal proportion of protected methods.

  H_{A4ii}: Classes at the top of an inheritance hierarchy have a greater proportion of protected methods than classes at the bottom.

This hypothesis is based on the belief that classes located at the top of the inheritance hierarchy should contain all generic behaviour, and the inheriting classes should have additional attributes and methods to satisfy their own additional functionalities.

4.4.2 Data collected

The data used to aid the investigation was collected automatically for three of the five C++ systems by parsing only the *header files* (Edge, ET++, Rocket). Since only paper versions of the GNU and LEDA systems were available at the time of the collection, data from the header files of these two systems was collected manually. Correspondingly, the data from the five Java systems was collected automatically by using Java software (see Appendix B).

We define a class as non-inheriting if it has neither a subclass nor a superclass. Since encapsulation controls the accessibility (private, protected and public) of class members, we chose metrics that were related to the declaration types for class attributes and methods. For each of the C++ and Java systems, the following metrics were collected:

1. The number of private attributes found in each class (see metrics 1 and 2 in Section 3.3.5, i.e., ***priPA*** and ***priNPA***).
2. The number of private methods found in each class (see metrics 4 and 5 in Section 3.3.5, i.e., ***priPM*** and ***priNPM***).
3. The number of protected attributes found in each class for classes which do not inherit in the system (cf. metrics 6 and 7 in Section 3.3.5, i.e., ***proPA*** and ***proNPA***).
4. The number of protected attributes found in each class for classes which inherit in the system (cf. metrics 6 and 7 in Section 3.3.5, i.e., ***proPA*** and ***proNPA***).
5. The number of protected methods found in each class for all classes which do not inherit in the system (cf. metrics 9 and 10 in Section 3.3.5, i.e., ***proPM*** and ***proNPM***).
6. The number of protected methods found in each class for all classes which inherit in the system (cf. metrics 9 and 10 in Section 3.3.5, i.e., ***proPM*** and ***proNPM***).
7. The number of public attributes found in each class (see metrics 11 and 12 in Section 3.3.5, i.e., ***pubPA*** and ***pubNPA***).
8. The number of public methods found in each class (see metrics 14 and 15 in Section 3.3.5, i.e., ***pubPM*** and ***pubNPM***).

9. The number of *friend* declarations found in each class (for C++ systems only).

In addition, for each of the five Java systems, we collected all the data (except for 9 which does not obtain) for classes which participate in inheritance at each level of the inheritance hierarchy, and for classes which do not participate in any inheritance hierarchy. We consider only class inheritance hierarchies; interfaces and abstract classes were not included in the inheritance hierarchies for data collection. We did not include interfaces in our analysis because they do not contain attributes and are not part of any inheritance hierarchy. Correspondingly, we did not include abstract classes because we consider them as a special type of class, some of whose methods may be left unimplemented – they are different to other classes in this respect.

Table 4.1 gives the breakdown in terms of the number of inheriting and non-inheriting classes found in each of the five C++ systems. We note that from the sample of classes chosen for the Edge system, more classes did not engage in inheritance than did. Of the remaining four systems, the framework ET++ system (Weinand et al. 1988) had the lowest proportion of non-inheriting classes (10.73%), whilst the highest proportion of non-inheriting classes belonged to GNU, the library-based system (37.86%). Although these figures only present samples of selected classes, they do reflect the kind of differences existing in the five C++ systems and, in particular with reference to the ET++ framework, which stood out from the rest in terms of its adherence to OO principles.

System	Sample Size	Inheriting	Non-inheriting
Edge	41	15	26
ET++	205	183	22
Rocket	209	172	37
GNU	103	64	39
LEDA	203	131	72

Table 4.1: The number of inheriting and non-inheriting classes for each of the five C++ systems

4.5 Data Analysis

In this section we provide the analysis and support for each of the four hypotheses we described in Section 4.4.1. We also discuss the role of *friends* in C++ classes with respect

to encapsulation and inheritance. We note that *n/a* refers to those cases where it is not applicable to calculate the normal approximation *p*-value and the *z* value, because of the non existence of data.

4.5.1 Hypothesis one

Table 4.2 shows the number of private methods and the total number of methods declared (which represents the sum number of private, protected and public methods) in inheriting and non-inheriting classes for each of the five C++ systems. The two library-based systems, GNU and LEDA, have a proportionately larger number of private methods vis-à-vis the other three systems and in particular with that of LEDA. One suggestion for the relatively high number of private methods in the two library-based systems may be due to the relatively large amount of care taken in the initial development of those classes, and subsequent fine-tuning which the same classes in these systems have undergone.

System	Inheriting		Non-inheriting	
	Private methods	Total number of methods	Private methods	Total number of methods
Edge	1	144	8	658
ET++	62	2310	4	196
Rocket	6	1601	6	226
GNU	57	546	45	856
LEDA	520	2952	242	1270

Table 4.2: The number of private methods and the total number of methods in inheriting and non-inheriting classes for each of the five C++ systems

Table 4.3 shows the p-values of the normal approximation test and the p-values of Fisher's exact test of the two proportions test (see Section 3.3.7.1.4) for private methods (as opposed to protected and public methods) in inheriting and non-inheriting classes for each of the five C++ systems. Since the data sets, in some cases (as in the number of private methods for inheriting classes of Edge and in the number of private methods for non-inheriting classes of ET++), are small, it is more accurate to rely on the results of Fisher's exact test.

System	z value	p-value	Fisher's p-value
Edge	-0.64	0.261	0.499
ET++	0.60	0.727	0.772
Rocket	-2.11	0.017	0.002
GNU	3.42	1.000	1.000
LEDA	-1.10	0.135	0.142

Table 4.3: The p-values of the two proportions test for private vs. total number of methods in inheriting and non-inheriting classes for the five C++ systems

We note that because we are running several tests, there is more danger of getting a false positive result. For this reason, the threshold p-value of 0.01 is used rather than 0.05 in terms of the Fisher's exact test results. In Table 4.3 the only p-value that is less than 0.01 belongs to the Rocket system. Therefore there is significant evidence to reject the null hypothesis at the 1% level for Rocket. For the remaining four systems (Edge, ET++, GNU and LEDA), the p-values of the Fisher's exact test are greater than 0.01, therefore we do not reject the corresponding null hypothesis (H_{01}). In other words, the proportion of private methods for inheriting classes does not differ from the proportion of the non-inheriting ones for four out of the five C++ systems under consideration.

We next consider the hypothesis that the proportion of private methods is *greater* (as opposed to protected and public) in inheriting classes than non-inheriting classes in C++ software systems. We apply the two proportions test to the data in Table 4.2, and again note that this hypothesis is not supported by the contents of Table 4.3a. Only for GNU was there significant evidence of having more private methods in inheriting classes than non-inheriting ones. In other words, no real pattern emerges in declaring private methods for inheriting or non-inheriting classes in the C++ software systems under consideration.

Opposite of alternative hypothesis one in the C++ systems
(The proportion of private methods (as opposed to protected and public methods) in inheriting classes is greater than those in non-inheriting classes)

System	z value	p-value	Fisher's exact p-value
Edge	-0.64	0.739	0.833
ET++	0.60	0.273	0.402
Rocket	-2.11	0.983	1.000
GNU	3.42	0.000	0.000
LEDA	-1.10	0.865	0.877

Table 4.3a: The p-values of the two proportions test for private vs. total number of methods in inheriting and non-inheriting classes for the five C++ systems

Based on the evidence presented, there is not enough support to accept H_{01}. Therefore, we conclude that it is not always the case that classes which do engage in inheritance have a lower proportion of privately declared methods than classes which do not engage in inheritance.

4.5.2 Hypothesis two

- The first sub-hypothesis:

Table 4.4 shows the number of protected attributes and the total number of attributes declared (representing the sum number of private, protected and public attributes) in inheriting and non-inheriting classes for each of the five C++ systems. Neither Edge nor ET++ has any protected attributes in their non-inheriting classes. On the basis that there is no other reason for having protected attributes other than for classes included in an inheritance hierarchy, it would seem, at first, that these two systems conform to OO principles.

System	Inheriting		Non-inheriting	
	Protected attributes	Total number of attributes	Protected attributes	Total number of attributes
Edge	0	38	0	256
ET++	266	615	0	58
Rocket	198	403	26	127
GNU	48	140	34	108
LEDA	20	350	15	383

Table 4.4: The number of protected attributes and the total number of attributes in inheriting and non-inheriting classes for each of the five C++ systems

The first possibility for the presence of protected attributes in the remaining three systems is that at some stage, those relevant classes were removed from the inheritance hierarchy without their attributes being re-defined from that of the protected category. Without knowing why these classes were removed from the hierarchy (i.e., the exact intentions of the original developers), it is difficult to hypothesise. However, one theory is that those classes were identified as *key* classes (Counsell et al. 2000) after they were placed in the inheritance hierarchy; they were thus removed to allow easier access to their attributes and methods. The second possibility (but less likely perhaps due to developer time and cost pressures) is that the developer anticipated incorporating those classes into the inheritance hierarchy at a later stage and hence declared attributes of relevant classes as protected at their creation.

Interestingly, Edge contains no protected attributes (Table 4.4) amongst the fifteen inheriting classes investigated (see Table 4.1) – no evidence of any protected attributes was therefore found for this system, suggesting that contrary to what was initially thought, it does not conform particularly well to OO design principles. However, one theory to explain

why no protected features were found in Edge may be that inheritance was used for pure reuse purposes and hence no protection was required. It is noted here that graphically-based systems do tend to use and need inheritance structures by their very nature (Harrison et al. 1998a).

One theory for why LEDA would contain such a relatively small number of protected attributes relates to the nature of classes in that system; classes in LEDA tend to be relatively small in terms of their number of methods (Counsell and Newson 2000), and revolve around the use of two or three attributes, a typical example being the manipulation of *x* and *y* co-ordinate values for a geometric object (declared as primitive, e.g., *int* or *float*). The GNU and Rocket systems were different to LEDA in this respect, as can be seen from the relatively high number of protected attributes for inheriting and non-inheriting classes.

From Table 4.4, it would seem that the protected definition of attributes for (non-) inheriting classes is used very sparingly and, in some cases, is not used at all. Moreover, evidence of protected attributes in classes not engaging in inheritance point to the existence of either *orphaned* classes, i.e., classes which were once part of an inheritance hierarchy but were removed at some point, perhaps because they were identified as potential *key* classes (central to the operation of the system as a whole); alternatively, it may be that the developers anticipated their use in another inheritance hierarchy at a later date. The latter explanation would also seem less plausible, since declaring attributes in classes as protected with a view to later inclusion in an inheritance hierarchy is more likely to confuse rather than assist another developer or software maintainer.

Table 4.5 shows the two proportions test statistics for each of the five C++ systems in terms of protected attributes (as opposed to private and public attributes) in inheriting and non-inheriting classes. For the five C++ systems, the p-values of the Fisher's exact test are greater than 0.01 and we do not therefore reject the null hypothesis (H_{02i}). We conclude that it is not always the case that classes which do engage in inheritance have a lower proportion of protected attributes than classes which do not engage in inheritance.

System	z value	p-value	Fisher's p-value
Edge	n/a	n/a	1.000
ET++	21.65	1.000	1.000
Rocket	6.57	1.000	1.000
GNU	0.47	0.680	0.726
LEDA	1.13	0.871	0.906

Table 4.5: The p-values of the two proportions test for protected attributes vs. total number of attributes in inheriting and non-inheriting classes for the five C++ systems

- The second sub-hypothesis:

Table 4.6 shows the number of protected methods and the total number of methods for inheriting and non-inheriting classes for each of the five C++ systems. Again, the Edge and ET++ systems show no evidence of protected methods in their non-inheriting classes, and the three remaining systems (as was found for attributes) contain varying levels of protected methods. Edge also shows no evidence of any protected methods in inheriting classes. For the Rocket system, the relatively high number of protected attributes, yet relatively low number of methods, is remarkable. This suggests one of two possible scenarios during the evolution of this system. Firstly, classes in the Rocket system, removed from the inheritance hierarchy, had their method declarations changed to private or public whilst the attribute declarations of classes remained unchanged. Secondly, those methods were initially defined in that way, namely, private or public (this latter scenario would seem unlikely, since data and behaviour are normally inherited in unison and hence we would expect both attributes and methods to be declared as protected, if this was indeed the case).

System	Inheriting		Non-inheriting	
	Protected methods	Total number of methods	Protected methods	Total number of methods
Edge	0	144	0	658
ET++	152	2310	0	196
Rocket	3	1601	1	226
GNU	33	546	25	856
LEDA	171	2952	12	1270

Table 4.6: The number of protected methods and the total number of methods in inheriting and non-inheriting classes for each of the five C++ systems

Table 4.7 shows the two proportions test statistics for each of the five C++ systems in terms of protected methods (as opposed to private and public methods) in inheriting and non-inheriting classes. For the five C++ systems, the p-values of the Fisher's exact test are

greater than 0.01 and we do not therefore reject the null hypothesis (H_{02ii}). We conclude that it is not always the case that classes which do engage in inheritance have a lower proportion of protected methods than classes which do not engage in inheritance.

System	z value	p-value	Fisher's p-value
Edge	n/a	n/a	1.000
ET++	12.76	1.000	1.000
Rocket	-0.56	0.287	0.411
GNU	2.67	0.996	0.998
LEDA	9.53	1.000	1.000

Table 4.7: The p-values of the two proportions test for protected methods vs. total number of methods in inheriting and non-inheriting classes for the five C++ systems

Finally, by inspecting Tables 4.7a and 4.7b, we see that the only system for which there is significant evidence of having consistently more protected features (attributes and methods) is ET++; this suggests that the ET++ user-interface framework system conforms best to OO practice.

Opposite of alternative hypothesis two (first sub-hypothesis) in the C++ systems
(The proportion of protected attributes (as opposed to private and public attributes) in inheriting classes is greater than those in non-inheriting classes)

System	z value	p-value	Fisher's exact p-value
Edge	n/a	n/a	1.000
ET++	21.65	0.000	0.000
Rocket	6.57	0.000	0.000
GNU	0.47	0.320	0.372
LEDA	1.13	0.129	0.167

Table 4.7a: The p-values of the two proportions test for protected attributes vs. total number of attributes in inheriting and non-inheriting classes for the five C++ systems

Opposite of alternative hypothesis two (second sub-hypothesis) in the C++ systems
(The proportion of protected methods (as opposed to private and public methods) in inheriting classes is greater than those in non-inheriting classes)

System	z value	p-value	Fisher's exact p-value
Edge	n/a	n/a	1.000
ET++	12.76	0.000	0.000
Rocket	-0.56	0.713	0.923
GNU	2.67	0.004	0.004
LEDA	9.53	0.000	0.000

Table 4.7b: The p-values of the two proportions test for protected methods vs. total number of methods in inheriting and non-inheriting classes for the five C++ systems

4.5.3 The role of friends

One feature of any analysis of encapsulation and inheritance which cannot be ignored in a study of this type relates to the use of *friends* in each of the five C++ systems. Friends

presents a violation of encapsulation, and a potential alternative to the use of inheritance. An earlier study (Counsell and Newson 2000) found that friends tended to be found at lower (i.e., deeper) levels of the inheritance hierarchies where they could take advantage of the inherited functionality. On a similar theme, an empirical study conducted by English et al. (English et al. 2005) examined the use of the *friend* construct and its effects on other OO measures (inheritance, coupling and size) across a large number of open-source software developed at least partially in C++. They found that classes which were declared as friends were bigger and had higher coupling than other system classes. Furthermore, they found no indication that friends are used as an alternative to inheritance even though a small number of the investigated systems tend to use friends as an alternative to multiple inheritance. In terms of the relationship between friends and encapsulation, they found that the relationship between the number of private and protected members in a class and the number of friends declared in the class was supported by 13 out of 21 systems under consideration.

System	Inheriting	Percentage	Non-inheriting	Percentage
Edge	12/15	80%	10/26	38.46%
ET++	32/183	17.48%	1/22	4.55%
Rocket	12/172	6.98%	0/37	0%
GNU	4/64	6.25%	11/39	28.21%
LEDA	69/131	52.67%	67/72	93.06%

Table 4.8: The number and percentage of friends (inheriting or non-inheriting classes)

Table 4.8 shows the proportion of friends found in each set of inheriting and non-inheriting classes together with the percentage that this represents. An interesting conflict can be seen between the three non-library based systems and the library-based systems GNU and LEDA. In the latter two, a higher percentage of friends was found in the set of non-inheriting classes than in the set of inheriting classes. For the other three systems, the opposite is true, namely, inheriting classes were found to have a higher percentage of friends than non-inheriting classes.

Friends may well have been added (or initially incorporated into classes) in the GNU and LEDA systems as a fast and effective substitute for other forms of coupling (including that of inheritance). While friends is generally considered to be poor programming practice, and has been associated with an increased incidence of faults (Briand et al. 1999b), they do have the advantage, nonetheless, of being simple and quick to introduce into code. Lack of

foresight at the design stage can easily be rectified at the implementation stage by the use of friends.

Table 4.9 shows the number of classes in each of the five C++ systems with at least one friend (inheriting and non-inheriting classes). Table 4.9 shows that the Edge system has the highest proportion of friends for inheriting classes, since graphically-based systems use inheritance for pure reuse; it seems that Edge developers used friends as an alternative to inheritance to avoid dependencies. It is also clear from Table 4.9 that the percentage of the non-inheriting classes with at least one friend is higher than the percentage of inheriting classes in the two library-based systems (GNU and LEDA).

System	Inheriting	Percentage	Non-inheriting	Percentage
Edge	5/15	33.33%	6/26	23.08%
ET++	21/183	11.47%	1/22	4.55%
Rocket	11/172	6.93.%	0/37	0%
GNU	4/64	6.25%	9/39	23.08%
LEDA	35/131	26.72%	44/72	61.11%

Table 4.9: The number of classes with at least one friend (inheriting or non-inheriting classes)

4.5.4 Hypothesis three

The two sub-hypotheses of Hypothesis three are discussed and analysed in tandem. Table 4.10 presents the number and the overall percentage of private attributes, the sum of private and public attributes, the number and the overall percentage of protected attributes, and the sum of protected and public attributes for all classes in each of the five Java systems.

System	Private attributes	Overall percentage	Sum of private and public attributes	Protected attributes	Overall percentage	Sum of protected and public attributes
GraphDraw	352	72.13	461	27	5.53	136
BSF	71	58.68	107	14	11.57	50
Libjava	176	49.30	356	1	0.28	181
Barat	268	65.21	305	106	25.79	143
Swing	1186	44.69	1795	859	32.36	1468

Table 4.10: The number and overall percentage of private attributes, the sum of private and public and the corresponding protected attributes data

Except for Libjava, the percentage values for private attributes are greater than those for public attributes (we note that the percentage of public attributes can be obtained by subtracting the sum of the private and protected percentages from 100). The percentage

values for private attributes, for three of the five Java systems, GraphDraw, BSF and Barat, are more than 58%, whilst those for the two library-based systems (Libjava and Swing) are less than 50%; this is perhaps due to the fact that library-based systems may need to provide more easy access to their features for their clients.

Looking at the percentage values for the protected attributes, for three systems (Graphdraw, BSF and Libjava) these values are relatively small. We observe that Libjava has the lowest percentage value for protected attributes, and the highest for public attributes.

The limited existence of protected attributes in GraphDraw, a graphically-based system, can be justified theoretically by the fact that inheritance may be used for reuse purposes only and therefore no protection was required (cf. Section 4.5.2). One suggestion for the difference between the two library-based systems (Libjava and Swing) in terms of the distribution of attribute declarations may be that Swing is a more mature library system; it has probably gone through more development iterations than Libjava.

Looking at the percentage values of the private and protected attributes for Barat and Swing in Table 4.10, it is evident that both systems have a conforming pattern with regard to the distribution of attribute declarations; that is, the proportion of private attributes is greater than that of public attributes and the proportion of protected attributes is greater than that of public attributes. These two systems (Barat and Swing) support both the alternative sub-hypotheses of Hypothesis three (H_{A3i} and H_{A3ii}), since the proportion of private or protected attributes is greater than that of public attributes.

It appears from Table 4.10, that in the three systems, GraphDraw, BSF and Libjava, developers prefer to declare attributes as public rather than protected. Remarkably, the Libjava system has the lowest number of protected attributes among the five Java systems; in fact, it has only one protected attribute.

We next offer some tentative explanations for this phenomenon.

Firstly, the possibility for the low presence of protected attributes in the GraphDraw, BSF and Libjava systems is that developers declared some attributes as public for the purpose of testing or maintenance, and because of time pressure they did not subsequently change the declaration to private or protected. Secondly, the explanation for having more public attributes than protected ones is that (perhaps) developers declared public attributes at key stages of the development process to provide new objects for ease of communication with existing objects. Alternatively, inheritance was used for pure reuse purposes and hence no protection was required, as is the case for the graphically-based system GraphDraw (cf. Section 4.5.2). Thirdly, the low number of protected attributes may be due to some key classes that contain a number of attributes intentionally declared as public in order to facilitate the access to these attributes from all classes in the system.

The Libjava system has the highest percentage value for public attributes among the five Java systems. The high percentage value for public attributes in Libjava is due to the class '*UnicodeBlock*' (see Appendix A), which contains 85 public attributes. By inspecting this class, we found it to be an inner class of the class '*Character*' (see Appendix A). Libjava does not support H_{A3i} as the other four systems do and as mentioned earlier, it has only one protected attribute. That attribute belongs to a class called '*SecurityManager*' (see Appendix A), which has most of the protected features (i.e., 7 protected methods) of the system.

In order to claim support for, or refutation of, Hypothesis three, the proportions of private or protected attributes (as opposed to public attributes) need to be re-assessed.

Table 4.11 shows the statistics of the one proportion test for the private attributes (as opposed to public attributes) in the five Java systems. It is clear from Table 4.11 that Libjava stands out from the other four systems. Statistics show that there is significant evidence (p-value < 0.01) for rejecting the corresponding null hypothesis (H_{03i}) for each of these four systems. In other words, in the four Java Systems (GraphDraw, BSF, Barat and Swing) the proportion of private attributes is greater than that of the public ones.

System	Sample p	95% Lower bound	Exact p-value
GraphDraw	0.763	0.728	0.000
BSF	0.664	0.581	0.000
Libjava	0.494	0.450	0.604
Barat	0.879	0.844	0.000
Swing	0.661	0.642	0.000

Table 4.11: Statistics of the one proportion test for private attributes vs. public attributes in the five Java systems

Similarly, Table 4.12 shows the statistics of the one proportion test for the protected attributes (as opposed to public attributes) in the five Java systems. From Table 4.12 statistics show that there is significant evidence (p-value < 0.01) for rejecting the corresponding null hypothesis (H_{03ii}) for only two Java systems (Barat and Swing). For the remaining three systems (Graphdraw, BSF and Libjava) the corresponding null hypothesis (H_{03ii}) is supported.

System	Sample p	95% Lower bound	Exact p-value
GraphDraw	0.199	0.144	1.000
BSF	0.280	0.178	1.000
Libjava	0.006	0.000	1.000
Barat	0.741	0.674	0.000
Swing	0.585	0.564	0.000

Table 4.12: Statistics of the one proportion test for protected attributes vs. public attributes in the five Java systems

After further study of Tables 4.11 and 4.12, we can see that, in terms of the alternative sub-hypotheses, H_{A3i} is supported by four of the five Java systems, whilst H_{A3ii} is supported only in the case of the Barat and Swing systems.

While there is significant support for the first sub-hypothesis of Hypothesis three, further investigations need to be carried out to determine whether to accept or refute this sub-hypothesis (H_{03i} and H_{A3i}). With respect to the second sub-hypothesis of Hypothesis three (H_{03ii} and H_{A3ii}) there is only limited support; we thus neither accept nor reject Hypothesis three. We conclude that the proportion of public attributes is not always the smallest compared to the corresponding private and protected proportions of attributes in OO software systems. This seems to tentatively indicate that software developers attend least to protected declarations.

4.5.5 Hypothesis four

In Java, the class 'Object' is the root of every class hierarchy, so every class has Object as a superclass. In the sequel, as a result of this assumption, we encounter multiple hierarchies. However, in this chapter, because we did not follow the inheritance hierarchy beyond the classes available in each system, we consider the level of inheritance for each class located at the top of its inheritance hierarchy as level zero, and so on.

Table 4.13 presents the number of attributes and methods, declared as private, protected and public at each level of the inheritance hierarchy and the number of classes (in parentheses) at each level of the inheritance hierarchy for GraphDraw. Table 4.13 shows that GraphDraw has only two levels of inheritance, comprising 6 classes. It can also be seen from this table that the majority of attributes and methods are at level zero. It is interesting to note that for inheriting classes in GraphDraw there are no protected attributes and methods. Even though the number of classes of this system is relatively small, GraphDraw has certain key classes containing large amounts of functionality. For example, the class '*Graph'* (see Appendix A) has 45 methods.

Inheriting classes at various levels		Private	Protected	Public
Attributes	Zero (2)	5	0	4
	One (4)	0	0	0
Methods	Zero (2)	6	0	48
	One (4)	0	0	31

Table 4.13: The number of private, protected and public attributes and methods at all levels of inheritance of GraphDraw

Regarding the BSF system, we found only one class '*EventAdapterImpl*' (see Appendix A) at level zero. This class (EventAdapterImpl) forms the root for 13 other classes and it only has one protected attribute and one public method at level zero. As can be seen from Table 4.14, all classes found at level one contain all the methods for BSF except one. No classes were found below level one for this system. Two possible reasons may account for this. Firstly, the design of BSF is such that developers should be able to tailor their classes to their own use without dependencies associated with levels of inheritance. Secondly, the BSF system may have been subject to less maintenance effort and, as a result, fewer classes have been added in order to enhance its functionality.

Inheriting classes at various levels		Private	Protected	Public
Attributes	Zero (1)	0	1	0
	One (13)	0	0	0
Methods	Zero (1)	0	0	1
	One (13)	0	0	31

Table 4.14: The number of private, protected and public attributes and methods at all levels of inheritance of BSF

Table 4.15 presents the data for the Libjava system. It is interesting to note that one of the classes, '*UnicodeBlock',* contains a large number of public attributes and is located at level one. Unexpectedly, there are no protected attributes at any level (a possible reason for this can be found in the second paragraph on page 93). Considering the number of classes at each level of inheritance in Table 4.15, we see that most of the methods are at levels zero, one and two. By inspecting the level four classes, we found that each of these classes had 2 or 3 public constructors and 1 attribute declared without an access specifier (package access). However, as we can see from Table 4.15, the latter classes have no methods at all; it may be that these classes make use of their superclasses. We note that the vast majority of classes are at levels two, three and four.

Inheriting classes at various levels		Private	Protected	Public
Attributes	Zero (3)	5	0	1
	One (4)	4	0	85
	Two (10)	2	0	0
	Three (18)	1	0	0
	Four (10)	0	0	0
Methods	Zero (3)	1	1	12
	One (4)	1	1	3
	Two (10)	2	0	4
	Three (18)	0	0	4
	Four (10)	0	0	0

Table 4.15: The number of private, protected and public attributes and methods at all levels of inheritance of Libjava

Similarly, Table 4.16 shows the corresponding data for attributes and methods for the Barat system. It is interesting to note that all levels of inheritance contain some number of attributes and methods, but Barat has the highest number of attributes and methods at level zero of the inheritance hierarchies. Again, we note that the vast majority of classes are at levels zero and one.

Inheriting classes at various levels		Private	Protected	Public
Attributes	Zero (10)	7	23	2
	One (13)	1	6	0
	Two (2)	0	2	0
Methods	Zero (10)	24	17	215
	One (13)	6	3	156
	Two (2)	2	0	2

Table 4.16: The number of private, protected and public attributes and methods at all levels of inheritance of Barat

Table 4.17 shows the data for the Swing system. It is clear that lower levels (three and four) of the inheritance hierarchies have fewer classes. (It is worth noting that levels zero and one contain the majority of attributes and methods.) In contrast to Libjava, all levels of the inheritance hierarchies of Swing contain some number of protected attributes (except level four) and methods, but both Libjava and Swing have the same DIT. As expected, large numbers of public attributes and methods were found at levels zero, one and two of the inheritance hierarchies. This can be accounted for by the nature of the Swing system as a library package.

Inheriting classes at various levels		Private	Protected	Public
Attributes	Zero (102)	265	367	224
	One (205)	221	128	34
	Two (48)	41	4	13
	Three (5)	5	3	0
	Four (1)	3	0	0
Methods	Zero (102)	102	652	1239
	One (205)	109	344	801
	Two (48)	3	34	113
	Three (5)	0	5	9
	Four (1)	0	1	3

Table 4.17: The number of private, protected and public attributes and methods at all levels of inheritance of Swing

Table 4.17 also shows large numbers of methods at the top of the inheritance hierarchies; the higher levels of the inheritance hierarchies contain high numbers of public methods; the opposite is true for lower levels. This may be due to inheritance subclassing employed for specialization which is the acknowledged use of inheritance, and for which design developers should strive (Budd 2002). We note that the vast majority of classes are at levels zero, one and two.

Table 4.18 shows the number and overall percentage of private, protected and public attributes and methods for classes that do not have any inheritance links (except for inheriting from abstract classes) for each of the five Java systems.

Non-inheriting classes (total number of classes)		Private attributes	Overall percentage	Protected attributes	Overall percentage	Public attributes	Overall percentage
Attributes	GraphDraw (43)	336	72.41	27	21.77	101	5.82
	BSF (45)	71	59.17	13	10.83	36	30.00
	Libjava (36)	164	63.08	1	0.38	95	36.54
	Barat (196)	252	70.59	74	20.73	31	8.68
	Swing (696)	651	48.36	357	26.52	338	25.11
Methods	GraphDraw (43)	112	27.59	1	0.25	293	72.17
	BSF(45)	27	10.04	11	4.09	231	85.87
	Libjava (36)	45	7.74	10	1.72	526	90.53
	Barat (190)	112	6.11	22	1.20	1698	92.68
	Swing (696)	333	7.34	409	9.02	3793	83.64

Table 4.18: The number and overall percentage of private, protected and public attributes and methods for non-inheriting classes for each of the five Java systems

From Table 4.18, it is clear that all the five Java systems have protected attributes and methods for the classes that do not belong to any inheritance hierarchy. However, these classes contain a significant number of protected attributes and methods in comparison to the classes that engage in inheritance. This may be because when developers created these classes they expected them to be used as superclasses in future development. The percentage values of public methods for the five Java systems are comparable and high; this conforms to expected practice in the declaration of methods in OO systems.

Comparing Tables 4.13 to 4.17 in terms of attributes, all the five Java systems have most of their private, protected and public attributes at levels zero and one. On the other hand, there are no protected attributes at all levels of inheritance in two out of the five Java systems (Tables 4.13 and 4.15); developers seem to prefer declaring attributes as private or public. For the remaining three systems (BSF, Barat and Swing) most of the protected attributes are found at the top of their inheritance hierarchies (Tables 4.14, 4.16 and 4.17).

We next consider methods; again, all the systems have a large amount of their methods at the first two levels of inheritance. Both BSF and GraphDraw have no protected methods at any level of inheritance. However, the majority of the protected methods for the other three systems (Libjava, Barat and Swing) are found at the top of their inheritance hierarchies.

Table 4.19 presents the percentage of inheriting and non-inheriting classes for each the five Java systems. Most of the system classes seem to be created in isolation, since the percentage of inheriting classes is overall low in comparison with that of the non-inheriting classes. Except for Libjava, the other four systems have more than 65% of their classes without inheritance links. This is perhaps because developers provide functionality without considering the available classes (due to time and cost pressures). Alternatively, they may prefer to create classes without support or links with other classes in order to avoid dependency problems.

System	Inheriting	Non-inheriting
GraphDraw	12.24%	87.76%
BSF	23.73%	76.27%
Libjava	55.56%	44.44%
Barat	11.31%	88.69.%
Swing	34.15%	65.85%

Table 4.19: Percentage of inheriting and non-inheriting classes in each of the five Java systems

All five Java systems have considerable numbers of protected attributes and methods, for the non-inheriting classes (Table 4.18); this may be because the developers' intention was to incorporate these classes in the inheritance structure in a future development.

With the purpose of claiming support for, or refutation of, Hypothesis four, the private, protected and public attributes and methods for classes at top levels of inheritance require re-assessment.

Table 4.20 shows the number of the protected attributes of classes located at the top levels of inheritance and the total number of protected attributes of classes located at all levels of inheritance for the five Java systems.

System	Protected attributes at the top levels of inheritance	Total number of protected attributes at all levels of inheritance
GraphDraw	0	0
BSF	1	1
Libjava	0	0
Barat	23	25
Swing	495	498

Table 4.20: The number of protected attributes at the top and all levels of inheritance for the five Java systems

Table 4.21 shows the statistics of the one proportion test for the protected attributes of classes that are located at the top levels of inheritance (as opposed to those at bottom levels of inheritance) in the five Java systems. Statistics from Table 4.21 show that there is significant evidence (p-value < 0.01) for rejecting the corresponding null hypothesis (H_{04i}) for Barat and Swing. For the remaining three systems (Graphdraw, BSF and Libjava), however, a conclusion cannot be drawn, in the case of GraphDraw and Libjava, because of the absence of protected attributes, and in the case of BSF support for the null hypothesis obtains.

System	Sample p	95% Lower bound	Exact p-value
GraphDraw	n/a	n/a	n/a
BSF	1.000	0.500	0.500
Libjava	n/a	n/a	n/a
Barat	0.920	0.769	0.000
Swing	0.661	0.642	0.000

Table 4.21: Statistics of the one proportion test for protected attributes located at the top levels of inheritance vs. those at the bottom levels in the five Java systems

Table 4.22 shows the number of the protected methods of classes that are located at the top levels of inheritance and the total number of protected methods of classes located at all levels of inheritance for the five Java systems.

System	Protected methods at the top levels of inheritance	Total number of protected methods at all levels of inheritance
GraphDraw	0	0
BSF	0	0
Libjava	2	2
Barat	17	19
Swing	996	1002

Table 4.22**:** The number of protected methods at the top and all levels of inheritance for the five Java systems

Table 4.23 shows the statistics of the one proportion test for the protected methods of classes that are located at the top levels of inheritance (as opposed to those at bottom levels of inheritance) in the five Java systems. Statistics from Table 4.23 show that there is significant evidence (p-value < 0.01) for rejecting the corresponding null hypothesis (H_{04ii}) for Barat and Swing; the null hypothesis (H_{04ii}) is accepted for Libjava. For the remaining two systems (Graphdraw and BSF), however, a conclusion cannot be drawn because of the absence of protected methods.

System	Sample p	95% Lower bound	Exact p-value
GraphDraw	n/a	n/a	n/a
BSF	n/a	n/a	n/a
Libjava	1.000	0.224	0.250
Barat	0.895	0.704	0.000
Swing	0.994	0.988	0.000

Table 4.23: Statistics of the one proportion test for protected methods located at the top levels of inheritance vs. those at the bottom levels in the five Java systems

After some further scrutiny of Tables 4.21 and 4.23, we can see that alternative sub-hypotheses H_{A4i} and H_{A4ii} are supported by two of the five Java systems (Barat and Swing). There is only limited support for the first and second sub-hypotheses of Hypothesis four; we thus neither accept nor reject Hypothesis four. Further investigations need to be carried out to determine whether to accept or refute these sub-hypotheses (H_{04i} and H_{04ii}). We conclude that the proportion of protected attributes for classes located at the top of the inheritance hierarchy is not always greater than the proportion of those located at the bottom of the inheritance hierarchy. Moreover, further investigations need to be carried out as to why a large number of the system classes, except in the case of Libjava, do not engage in inheritance.

4.6 Discussion

In any study of this nature, the caveats to its validity have to be considered. The first caveat which needs to be considered is that the five C++ systems investigated were all of similar type and size. This is counterpoised by the fact that the five C++ and the corresponding five Java systems were deliberately chosen to reflect a wide range of application domains and sizes (see Section 3.3.3).

The study could be criticised for only analysing the header files of the five C++ systems. However, it is unlikely, from a statistical point of view, that the results would differ had all the classes in each system been analysed.

In terms of the analysis of *friends* in each of the five C++ systems, the study could be criticised for treating each friend declaration as the same. It is possible that friends are used (particularly in the library-based systems) for the purpose of *operator overloading*, and not

as a means of accessing the class features of the class they are declared in. However, treatment of this possibility would be a matter for future research. We further accept that use of friends in C++ software may be a necessary evil (neither Java nor Smalltalk have an equivalent of the 'friends' facility); we could therefore blame the designers of the C++ language for their use. This, however, does not invalidate the results from this study.

Three themes, related to the five C++ systems, come across clearly from this chapter. The first theme is that there are significant amounts of protected class features in systems for classes which do not engage in inheritance. This can either be an encouraging sign (i.e., the developers deliberately made those features protected for future use), or a worrying sign, in the sense that those classes once participated in an inheritance hierarchy and no longer do; as such, their declarations should have been changed when it was established that they no longer participated in an inheritance hierarchy, for the benefit of future maintainers.

The second theme relates to the consistent results for the ET++ framework, which showed itself to conform well to OO encapsulation design principles. It contained zero protected declarations for classes not engaging in inheritance (Tables 4.4 and 4.6). It also tended to use protected declarations in inheriting classes liberally and had the second lowest number of violations of encapsulation via the *friends* facility (see Table 4.8). The research herein might point to the use of frameworks as a means of ensuring some level of design and implementation (architectural) stability. The ET++ system compares favourably with the two library-based systems, GNU and LEDA.

The third theme relates to the relatively high level of private methods in the two library-based systems, which we attribute to fine-tuning of the classes in the respective systems. This would imply that declaring class features as private is largely achieved after careful thought has been given to the nature of the methods in a class.

Correspondingly, two themes emerge clearly from studying the five Java systems in this chapter. The first theme is that the existence of protected features is limited in most of the Java systems, especially for inheriting classes. This may be a sign that developers find it easier to declare class features either as private or public instead. It also draws attention to

the need for solutions to these symptoms; that is, the existence of public attributes and the scarcity of protected ones.

The second theme is the limited use of inheritance between concrete classes for non-library-based systems (GraphDraw, BSF and Barat). The depth of the inheritance tree in these systems is less than that for the remaining two library-based systems. At the same time, the number of inheriting classes in GraphDraw, BSF and Barat is significantly less than the number of non-inheriting classes. These observations may be an indication that there is a need for some type of refactoring, especially related to tidying up the inheritance structure, such as *pull up field/method*, *extract subclass*, and *extract superclass* (Fowler 2000).

4.7 Summary

In this chapter we have investigated encapsulation trends and the relationship between encapsulation and inheritance in ten real-life software systems. We examined the trends of declaring attributes and methods as private, protected or public in classes of five C++ and five Java systems. We found that, to an extent, encapsulation has not been properly used when inheritance is considered. Moreover, declaring attributes as public, which violates the principle of encapsulation, appears to exist in some of the investigated systems.

The OO paradigm implements private, protected and public access specifiers as part of its syntax. As a fundamental OO concept, encapsulation was incorporated into programming languages to provide the developer with a means by which access to the private features of a class could be controlled (or encapsulated) and public features made available to all classes. The responsibility for enforcing these access mechanisms was thus devolved to the compiler and freed the developer from associated implementation and testing issues.

The concept of inheritance is also a core feature of OO languages because it promotes the reuse of code and, in theory, reflects the way that we as humans structure and manipulate information. Proper use of inheritance requires the careful consideration of how class features should be encapsulated. While public features can be shared by any other class, the

protected keyword relates specifically to the use of inheritance by allowing only subclasses of a class X to access the protected class features of X.

Of course, the *friends* facility (as a form of coupling) violates encapsulation because it allows access by a class, say Y, to the private parts of class X. Friends can thus be used within an inheritance hierarchy to take advantage of all superclass functionality, whether private, protected or public. While this is a convenient mechanism for developers, the encapsulation/inheritance semantics are completely subverted as a result. (We do accept that friends have a valid purpose when used for operator overloading, but in the context of the Thesis, we think of 'friends' usage in the former sense.)

The use of friends stores up future maintenance difficulties, since we can no longer rely on the compiler to prevent side-effects. Frequent misuse of friends in this sense, together with other inappropriate use of encapsulation, thus leads to a spiral of poor maintenance, anomalies and 'decay' in code, which can only be addressed by techniques such as refactoring. In Chapters 5 and 6, we thus explore the possibilities and potential for remedying encapsulation and inheritance anomalies caused over time by poor developer maintenance practice.

In the next chapter, we empirically investigate the potential for the 'Encapsulate Field' refactoring whose mechanics change an attribute defined as 'public' to that of 'private'.

CHAPTER 5 Encapsulate Field Refactoring

5.1 Introduction

Encapsulation is a key principle of the OO paradigm (Snyder 1986). Equally, proper use of inheritance requires us to consider protected declarations as standard design practice. One of the techniques widely used to improve the structure and comprehension of software systems is refactoring. We found in the previous chapter that encapsulation had not been properly used when inheritance was considered and declaring attributes as public only appears to exist in some of the investigated systems.

In this chapter, we empirically investigate the opportunities, benefits, and problems of refactoring related specifically to encapsulation. A sample of classes from each of the five Java systems, which are described in Chapter 3, was chosen and the 'Encapsulate Field' (EF) refactoring was then considered (Fowler 2000). The EF refactoring changes public attribute declarations to private thereby providing added encapsulation. Empirical results indicated several key reasons why this particular refactoring, while trivial in principle, can be either simple or difficult to implement in practice, depending on the features of the class and its role within the inheritance hierarchy. The problem of compound refactorings (requiring the application of more than one refactoring, e.g., 'move method') is in turn dependent on the ease of carrying out 'pre-refactorings'. We support our analysis with reflection on Fowler's original refactorings and empirical evidence from the five Java systems.

This chapter is organised as follows: in the next section, motivation and related work is described, followed in Section 5.3 by a description of EF refactoring. Details of the empirical evaluation are described in Section 5.4. Section 5.5 provides the data analysis for the research. Finally, a summary of this chapter is given in Section 5.6.

We note that part of the research on which this chapter is based was first published in (Najjar et al. 2005).

5.2 Motivation and Related Work

A key motivation for the work described in this chapter relates to earlier work (Counsell et al. 2002), as presented in Chapter 4, where we empirically investigated the trends in encapsulation and inheritance from five C++ and five Java systems. It was found therein that only one system (a framework – ET++ (Weinand et al. 1988)) conformed to sound encapsulation principles in terms of the distribution of private, protected and public attributes and methods; in each of the remaining four C++ systems, protected attributes and methods in classes not using inheritance were found; this would suggest that over time, the declaration of attributes tends to deteriorate in many cases contrary to the fundamental principles of OO. Similarly, the existence of protected features is limited in most of the five Java systems, especially for inheriting classes; it seems that it was easier for developers to declare attributes as public instead of protected.

Our study in this chapter draws attention to the importance of the inheritance hierarchy with reference to the five Java systems, and the subsequent need for refactoring in order to ameliorate the effects of encapsulation anomalies. In Snyder (Snyder 1986), the relationship between encapsulation and inheritance was studied; it was suggested that introduction of inheritance severely compromised the benefits that encapsulation offered. On a similar theme, an experiment, conducted by Skoglund (Skoglund 2003), used software engineers and their view of encapsulation issues. Some of the interviewed subjects stated that, because of time pressure and testing reasons, they often changed private declarations to public. This type of study also motivates the research described in this chapter, namely, in order to overcome encapsulation anomalies we change public declarations to private.

In terms of refactoring research, work has been done on investigating the benefits (both empirically and theoretically) gained from applying the techniques of refactoring (Opdyke 1992, Demeyer et al. 2000, Tokuda and Batory 2001, Mens and Tourwe 2004, O'Keeffe

and O'Cinne'ide 2006). In Najjar et al. (Najjar et al. 2003), the opportunities, benefits and problems of refactoring class constructors across a sample of classes from the same five Java systems as used herein were investigated. The refactoring 'Replace Multiple Constructors with Creation Methods' proposed by Kerievsky (Kerievsky 2004) was applied. Benefits in terms of improved encapsulation and for potentially reduced numbers of comment lines were found. Potential improvement in class comprehension was also a feature of that refactoring.

Finally, a key feature of our analysis is the use of metrics to quantitatively capture the features of the systems under investigation. Many metrics have been proposed and used for analysing OO software both theoretically and empirically (Chidamber and Kemerer 1994, Lorenz and Kidd 1994, Harrison et al. 1998b, Briand et al. 1999b, Cartwright and Shepperd 2000).

In this chapter, we use simple counts of the number of the class feature 'number of attributes'. We also include counts of the distribution of classes in the inheritance hierarchies of the five Java systems. One feature of the systems studied is the set of dependent classes for a particular class. Each dependency could be seen as a coupling and some research has also addressed this area (Briand et al. 1997b, Harrison et al. 1998a); a sound SE tenet is that developers should aim to minimise coupling in their code (Pressman 2000).

5.3 Encapsulate Field Refactoring

In this chapter, we empirically investigate the opportunities, benefits and problems associated with applying the EF refactoring, first proposed by Fowler in his seminal text (Fowler 2000). The EF refactoring changes the declaration of an attribute from public to private; accessors are then provided to offer access to this attribute to classes which had free access before. According to Fowler, a principal tenet of OO is that all data should be made private. When data is declared public however, other objects can access and modify that data without the owning object knowing. There are, therefore, implications for ease of

maintenance and program complexity, i.e., to maximise maintainability and minimise program complexity, the developer should take advantage of the in-built protective features offered by the Java language.

In order to apply EF refactoring, according to Fowler (Fowler 2000), five steps should be followed:

- Create getting and setting methods.
- "Find all clients outside the class that reference the field. If the client uses the value, replace the reference with a call to the getting method. If the client changes the value, replace the reference with a call to the setting method. *If the field is an object and the client invokes a modifier on the object, that is a use. Only use the setting method to replace an assignment.*" (Fowler 2000).
- Compile and test after each change.
- When all references have been modified, change the attribute declaration to private.
- Compile and test.

After each step a rigorous test should be made to ensure that the external behaviour of the software system is preserved. For any type of refactoring, testing after each step in order to guarantee that the external behaviour of the software system has been preserved, is of paramount importance. We have chosen the class *DPoint3* from GraphDraw as an example of EF refactoring.

5.3.1 Example: the DPoint3 class

The class DPoint3 has three public attributes which are used by two other classes (dependent classes) *EdgePropertiesDialog* and *Node.*

```
public class DPoint3
{
     …
     public double x, y, z;
     …
} // end of DPoint3 class
```

Statement number 8 in the EdgePropertiesDialog class has direct access to the three public attributes, x, y and z.

```
1.       public class EdgePropertiesDialog extends Dialog
2.       {
3.       …
4.       public void setEdge(Edge edge_in, Graph graph_in)
5.            {
6.            …
7.            DPoint3[] points = edge_.points();
8.            for(int i = 0; i < points.length; i++)points_string +=
                   points[i].x + " " + points[i].y + " " +
                   points[i].z + "\n";
9.            pointsText_.setText(points_string);
10.           …
11.           }
12.      …
13.      }//end of EdgePropertiesDialog class
```

Also, in the Node class the two statements, 9 and 10, have direct access to the two public attributes, x and y.

```
1.       public class Node implements Cloneable, ImageObserver
2.       {
3.       …
4.       public void draw(Component comp, Graphics graphics, Matrix44
              transform, int quality)
5.            {
6.            …
7.            DPoint3 position = new DPoint3(x_, y_, z_);
8.            position.transform(transform);
9.            int x = (int)position.x;
10.           int y = (int)position.y;
11.           …
12.           }
13.      …
14.      }//end of Node class
```

The dependent classes, EdgePropertiesDialog and Node, no longer have direct access to the public attributes, x, y and z, of DPoint3. To restore direct access set and get methods will be used for each of the DPoint3 public attributes.

First we have to create the set and get methods for the public attributes of the DPoint3 class (x, y and z), namely

```
public void setx(double d) { return (x = d);}
public void sety(double d) { return (y = d);}
public void setz(double d) { return (z = d);}
```

```
public double getx() {return x;}
public double gety() {return y;}
public double getz() {return z;}
```

In the EdgePropertiesDialog class, statement 8 will be replaced by the statement

```
points_string += points[i].getx() + " " +
points[i].gety()+" " + points[i].getz() + "\n";
```

while in the Node class statements 9 and 10 will be replaced by

```
int x = (int)position.getx();
int y = (int)position.gety();
```

After each of the changes, we have to compile and test to ensure that the external behaviour of the classes has been preserved.

In order to investigate the EF refactoring, samples of classes were chosen from the five Java systems and the potential for applying the mechanics of this refactoring were investigated. Results showed certain potential for applying the refactoring *per se*. In other words, no shortage of opportunities was found for applying the refactoring (public attributes were found in a number of classes in each system). However, three features were exhibited by the five Java systems that suggest applying the EF refactoring is not as straightforward or applicable as it first seems. Firstly, the number of dependent classes requiring changes as a result of applying the EF refactoring may prohibit the refactoring, secondly, the large number of classes with zero attributes would seem to render this refactoring almost redundant.

A final finding was the practical trade-off and applicability of the EF refactoring when considering different application domains. Some of the systems studied were found to be more amenable to the EF refactoring than others. This chapter also raises a number of current issues in the refactoring field; these are considered in Sections 5.5.1 and 5.5.2.

5.4 Empirical Investigation

The empirical investigation is divided into two parts. Firstly, we present the salient features of data collection. Secondly, we provide the analysis of the data in terms of dependent classes and zero attributes in the context of inheritance.

5.4.1 Data collected

A software tool was developed (see Appendix B) to automatically collect the required data from the five Java systems; the following data was then collected:

1. The total number of attributes (private, protected and public) declared in each class.
2. For each class in the sample, the number of classes that 'used' those public attributes. Such classes are termed 'dependent classes'.

For the latter category of data collected, we interpret 'used' as all those classes which reference and/or modify any attributes under consideration. Part of the mechanics of the EF refactoring proposed by Fowler (Fowler 2000) is to "find all clients outside the class that reference the field" and then provide getting and/or setting methods to provide access to those clients. Fowler also states that, as part of the mechanics of the EF refactoring,

> *"If the field is an object and the client invokes a modifier on the object, that is a 'use'. Only use the setting method to replace an assignment".*

The mechanics of the EF refactoring can therefore become even more involved if the type of an attribute being considered is an object rather than a *primitive* type such as 'int' or 'boolean'. Herein, we consider a 'use' as any occurrence of an outside object accessing and/or changing an attribute of the class being considered.

In addition to the metrics collected, we also considered, for each class, its position in the inheritance hierarchy, where Object is at the root of the entire hierarchy. Note that we did not include interfaces or abstract classes in our study. Interfaces were excluded because

they make all of their defined attributes and methods automatically public. On the other hand, we did consider abstract classes as a special type of class some of whose methods may be left unimplemented - they are different to other classes in this respect.

5.5 Data Analysis

Our analysis of the five systems revealed, in each case, a number of anomalies associated with the EF refactoring. The first of these relates to the dependencies of other classes (i.e., clients) when implementing the EF refactoring. In other words, the occurrence of 'uses' by objects of the set of attributes being considered.

Table 5.1 presents the total number of classes and the numbers of sample classes which were taken as a basis for our investigation in this chapter (see Section 3.3.3.1).

System	Total	Classes (more than two attributes)	Sample Size
GraphDraw	49	41	9
BSF	59	19	4
Libjava	81	19	4
Barat	221	64	13
Swing	1043	312	63

Table 5.1: Number of classes with two or more attributes for the five Java systems

5.5.1 Dependent classes

We observe that dependencies between classes present 'coupling'. We view any coupling, particularly dependencies which, in some sense violate sound OO practice, as an overhead. Having to establish which classes 'use' the public attributes of another class is a heavily time-consuming and computationally costly activity. Table 5.2 shows the sample of the nine classes taken from GraphDraw. It is clear that for the five classes with public attributes, on three occasions (Classes 5, 6 and 9) at least one other class uses at least one of the public attributes. As such, each of those three classes would have to be modified in some way to preserve access to the public attribute(s) which would now have become private.

GraphDraw System				
Class	Attributes	Private	Public	Dependent Classes
1	2	2	0	n/a
2	3	0	3	0
3	4	4	0	n/a
4	6	6	0	n/a
5	12	0	12	1
6	13	3	6	4
7	20	13	2	0
8	30	30	0	n/a
9	66	57	9	1

Table 5.2: Attributes and dependent classes for GraphDraw

Additionally, the classes containing attributes whose declaration had been changed from public to private also need to provide accessors (i.e., setters and getters) for each attribute. The EF refactoring is thus problematic for these three classes.

We note that the *n/a* entry in the table appears when there are no public attributes in the class (and hence no dependent classes). Furthermore, we note that the number of protected attributes in the class can be derived by subtracting the sum of private and public attributes from the total number of attributes.

Table 5.3 shows the same data for the BSF system. Only one class has potential for applying the EF refactoring, but five classes need to be modified in this instance. This highlights a controversial yet interesting aspect of refactoring, generally. The time required to modify and test those six classes (that includes the class with the attribute being modified) presents an opportunity cost. In other words, other refactorings may be perceived as more viable, more beneficial and may require less time and effort. Equally, it may be more worthwhile choosing classes whose dependencies are fewer, if only limited resources for refactoring and testing time are available to the developer. One of the reasons that Fowler (Fowler 2000) states for developers not doing refactoring as much as they should is simply lack of time.

BSF System				
Class	Attributes	Private	Public	Dependent Classes
1	2	2	0	n/a
2	3	0	3	5
3	7	7	0	n/a
4	25	25	0	n/a

Table 5.3: Attributes and dependent classes for BSF

This limited opportunity for the EF refactoring comes as no surprise, since in previous work using the same five Java systems, BSF consistently showed itself to conform best to OO design principles (Najjar et al. 2003). Thus we would have expected to find little opportunity for the EF refactoring in BSF.

Interestingly, Table 5.3 also shows the classes chosen to contain zero protected attributes, where as mentioned earlier the number of protected attributes can be obtained by subtracting the sum of private and public attributes from the total number of attributes; this would seem to make sense for a framework-based system such as BSF where design decisions are left to the developer. Yet more interesting is why it is there are *any* public attributes if the generally accepted tenet is for attributes to be declared private? Again, we can only hypothesise that it is left to the developer to tailor the particular attributes to their own requirements and the developers of BSF were judging the likely role that the attributes would play when they are used as part of an application.

Table 5.4 shows the analogous data for Libjava. Most remarkably, no classes from the sample chosen have any dependencies; this would suggest that an EF refactoring would be considerably more easily achievable than for the previous two systems. Again, this highlights another current refactoring issue; specific refactorings may be more appropriate to different types of application domain. The Libjava library-based system appears to be a system amenable to the EF refactoring and every class in the sample is a candidate for refactoring.

Libjava System				
Class	Attributes	Private	Public	Dependent Classes
1	4	1	3	0
2	5	4	1	0
3	8	2	6	0
4	112	110	2	0

Table 5.4: Attributes and dependent classes for Libjava

Table 5.5 shows the corresponding data for the Barat system. Four of the classes have public attributes and are hence candidates for the EF refactoring. However, only two of the four classes have zero dependencies. These two classes provide scope for refactoring.

Barat System				
Class	Attributes	Private	Public	Dependent Classes
1	2	2	0	n/a
2	2	2	0	n/a
3	2	2	0	n/a
4	2	1	0	n/a
5	3	3	0	n/a
6	3	0	0	n/a
7	4	4	0	n/a
8	4	0	0	n/a
9	5	3	2	2
10	6	0	0	n/a
11	8	0	8	3
12	14	8	4	0
13	21	5	2	0

Table 5.5: Attributes and dependent classes for Barat

The largest data sample was that for the Swing system. Table 5.6 shows the data for the Swing system. Despite the relatively large sample size of the system, the number of candidate classes for EF refactoring is small; only 11 classes have public attributes from a sample size of 63; the minimum of public attributes is 1 and the maximum 80. We also note that 13 classes have zero private and public attributes. The number of class dependencies in this system is relatively small, namely, 11. Out of these only 4 are most appropriate for the EF refactoring, namely, those that have an entry of zero under the column 'Dependent Classes'.

An interesting feature in Swing is the existence of a small set of classes with large numbers of public attributes. We would consider these classes as key classes (Counsell et al. 2000). We view a key class as one with large numbers of attributes and methods. As such, their existence is of importance to the overall functioning of the system (many classes may depend on key classes).

Inspection of the samples of classes chosen revealed the Libjava system to comprise a class, called '*StrictMath*', with 112 attributes and 38 methods. Equally, GraphDraw has a class, called '*GraphCanvas*', with 66 attributes and 63 methods. We suggest that it is not always true that these types of classes are obvious classes for refactoring. In many cases, the classes have large numbers of attributes and methods for a valid reason.

Swing System				
Class	Attributes	Private	Public	Dependent Classes
1	2	2	0	n/a
2	2	0	0	n/a
3	2	0	0	n/a
4	2	0	0	n/a
5	2	0	0	n/a
6	2	2	0	n/a
7	2	2	0	n/a
8	2	2	0	n/a
9	2	2	0	n/a
10	2	1	1	3
11	2	0	0	n/a
12	2	2	0	n/a
13	2	2	0	n/a
14	2	2	0	n/a
15	3	3	0	n/a
16	3	2	0	n/a
17	3	3	0	n/a
18	3	0	0	n/a
19	3	0	0	n/a
20	3	3	0	n/a
21	3	3	0	n/a
22	3	3	0	n/a
23	3	1	0	n/a
24	3	3	0	n/a
25	3	3	0	n/a
26	4	0	0	n/a
27	4	0	0	n/a
28	4	1	0	n/a
29	4	4	0	n/a
30	4	4	0	n/a
31	4	0	0	n/a
32	4	0	0	n/a
33	4	4	0	n/a
34	5	0	2	4
35	5	3	0	n/a
36	5	5	0	n/a
37	5	0	0	n/a
38	5	5	0	n/a
39	5	5	0	n/a
40	6	6	0	n/a
41	6	0	6	1
42	7	0	2	5
43	7	7	0	n/a
44	7	5	0	n/a
45	8	2	0	n/a
46	8	8	0	n/a
47	9	8	0	n/a
48	9	0	9	0
49	10	0	0	n/a
50	10	10	0	n/a
51	11	1	0	n/a
52	12	6	1	1
53	13	11	0	n/a
54	13	12	0	n/a
55	15	8	0	n/a
56	17	2	4	0
57	20	20	0	n/a
58	23	4	0	n/a
59	28	26	2	0
60	31	5	0	n/a
61	40	9	16	0
62	53	23	29	2
63	81	1	80	14

Table 5.6: Attributes and dependent classes for Swing

In the next section, we consider another feature of the five Java systems studied which renders EF refactoring inapplicable, namely, that of classes with zero attributes and the role that the inheritance hierarchy plays in this instance.

5.5.2 Zero attributes and inheritance

The purpose of EF refactoring is to modify the declaration of attributes from public to private. EF thus assumes that the classes do indeed have attributes. Table 5.7 represents the number of classes in each system with zero attributes; it also shows the total number of classes in each system and the percentage of that total which classes with zero attributes represents. Clearly, a high percentage of classes with zero attributes renders the EF refactoring inapplicable in these cases. For the BSF, Libjava and Swing systems, over fifty percent of classes fall into this category and well over a third of classes in the Barat system. This feature raises another interesting issue in refactoring. The lack of opportunity for carrying out this refactoring might at first seem to be a disadvantage. However, the opposite is really the case. Other refactorings such as the 'Move Method' (MM) (Fowler 2000), where one method is moved to a class more in line with its coupling links, can be applied without any requirement on the part of the developer to look for and preserve dependencies if there are zero attributes. In fact, Fowler states that

"Once I've done Encapsulate Field I look for methods that use the new methods to see whether they fancy packing their bags and moving to the new object with a quick Move Method."

In other words, the absence of any opportunity to carry out the EF refactoring makes it easier to undertake certain other types of refactoring. Earlier work by Counsell et al. (Counsell et al. 2003) has shown empirically that manipulating methods is one of the most common refactorings over different versions of the same software. The lack of opportunity for one refactoring may thus pave the way for other refactorings to be applied.

Another interesting feature of Table 5.7 is the high percentage of zero classes for the two library-based systems, namely, Libjava and Swing. One explanation for these high percentages might be that the classes with zero attributes inherit the data they require from

superclasses and do not need attributes and methods of their own. They do not therefore tend to have many attributes of their own.

System	Total	Classes (with zero attributes)	Percentage
GraphDraw	49	5	10.20
BSF	59	34	57.62
Libjava	81	52	64.20
Barat	221	76	33.93
Swing	1043	604	57.14

Table 5.7: Number of classes with zero attributes in each system

Table 5.8 illustrates a further interesting feature of the five Java systems. It shows how few classes with zero attributes are located in the middle of the inheritance hierarchy, when they are expressed as a percentage of the total number of Classes with Zero Attributes (CZA).

System	Classes located in 'middle' of inheritance hierarchy	Percentage of CZA	Leaf classes	Percentage of CZA
GraphDraw	0	n/c	5	100
BSF	0	n/c	34	100
Libjava	8	15.38	44	84.62
Barat	3	3.95	73	96.05
Swing	27	4.47	577	95.53

Table 5.8: Pattern in distribution of classes in the inheritance hierarchy

Here, we define a class in the middle of the inheritance hierarchy as one which is neither a root nor a leaf class. We define a root class as one extending directly or indirectly the universal class 'Object'. It also shows that the vast majority of classes with zero attributes are leaf classes, i.e., classes with no descendants. In other words, those classes are more likely to inherit (i.e., to extend other classes) than be inherited (extended) themselves by other classes.

From a refactoring perspective, it thus becomes easier to carry out MM refactoring if the class being considered has no other classes inheriting from it, since we have fewer dependencies to check. On the other hand, if the inheriting class makes use of features in a superclass, then the MM refactoring is more problematic.

Table 5.8 also shows the percentage of CZA that the number of leaf classes represents. We remark that the percentages are exceptionally high for all systems. Frustratingly, we can

offer no explanation for the very low numbers of classes in the middle of the inheritance hierarchy and view this as an investigation and potential avenue for future research. It may well be symptomatic of the emphasis placed on inheritance in Java.

5.6 Summary

In this chapter, we have empirically investigated the potential for applying the EF refactoring (Fowler 2000). The main conclusion of this study is that applying the EF refactoring is often problematic because of the dependencies of other classes on those attributes whose declaration is changed from public to private; it is also inapplicable in many cases due to lack of public attributes. Finally, and paradoxically, the traits in the inheritance hierarchy make it easier to perform the MM refactoring. Failure to find and apply some refactorings is actually an advantage in terms of what it allows the refactorer to achieve.

In the next chapter we pursue further the empirical investigation of refactoring whereby we replace multiple constructors with creation methods.

CHAPTER 6 Refactoring Class Constructors

Constructors play an essential role in OO languages as a means of object creation. Yet, very little empirical evidence exists on constructors, trends in their composition and how they impact comprehension and encapsulation of OO classes.

In Chapter 5 we presented an empirical study describing the benefits, problems and opportunities associated with the EF refactoring, originally introduced by Fowler (Fowler 2000). In this chapter, we empirically investigate the opportunities, benefits and problems of refactoring class constructors across a sample of classes from the same five Java systems. The refactoring used, namely, RMCCM, was applied to each of a set of classes containing three or more constructors. Empirical results showed benefits in terms of removed (duplicated) lines of code across the majority of systems. They also showed the potential for improved class comprehension by the creation of non-constructor methods (as a replacement for constructors) and improved encapsulation of class features through use of a private catchall constructor.

In terms of problems encountered, frequent and inconsistent use of the *super* construct made refactoring prohibitively difficult in some cases; the existence of Java interfaces also meant a lack of scope for constructor refactoring. We also investigated the role that inheritance played in the choice of classes to refactor as well as patterns in comment lines among the constructors studied. Results overall indicate a clear and tangible benefit to be gained from investigation and implementation of refactoring techniques in Java, but with caution being exercised in certain cases. Refactoring of constructors in practice is not as straightforward as the theory might suggest and a number of factors need to be considered before such refactoring is undertaken.

We note that part of the research on which this chapter is based was first published in (Najjar et al. 2003).

6.1 Introduction

In the OO paradigm, *constructors* are functions which assign and validate the initial values of the class features of the object being created. Constructors differ in many respects from ordinary functions; they do not have a return type, they share the same name of the class in which they are defined and they cannot be inherited when they are part of an inheritance hierarchy. In a language like C (Kernighan and Ritchie 1978), faults may arise because a developer forgets to initialise a variable (Ostrand et al. 2004). This problem is partially averted in OO languages, but the trade-off in a class with multiple constructors is that it is often difficult to tell the purpose of each constructor when they differ in only minor respects.

In this chapter, we investigate the opportunities, potential benefits and problems from refactoring classes with a large number of constructors. The five Java systems described in Chapter 3 were empirically investigated with respect to the RMCCM refactoring (Kerievsky 2004). The underlying principle of this refactoring is that transformation of constructors into normal methods improves the developer's ability to understand the class (under consideration) as a whole, saves lines of code duplicated across multiple constructors and aids encapsulation by the creation of a single private catchall constructor (where previously the constructors were all defined as public).

Results from our empirical study showed that in terms of removed (duplicated) lines of code, there are clear benefits to be gained. The potential for improved class comprehension by the creation of non-constructor methods should, in theory, make the task of developers easier, because they do not necessarily need to understand what each of a set of constructors does. Since non-constructor methods created from the corresponding constructors will usually be named according to their function, it should be much easier for the developer to identify the appropriate method which invokes the constructor. The empirical findings were not all supportive of the refactoring process. In certain cases, inconsistent use of the super construct for invoking the superclass constructor caused problems with assessing the opportunity for the refactoring process. On the other hand, the position of candidate classes (for the said refactoring) in the inheritance hierarchy and the

habits of developers in terms of comment lines may be influential and/or supportive factors in deciding when to carry out this type of refactoring.

The layout of this chapter is as follows. In the following section, motivation and related work are presented. In Section 6.3 we give details of the refactoring and the data collected from the sample of classes in each Java system. In Section 6.4 we present the results of our empirical study. We then show, in Section 6.5, how nuances of the Java language can cause problems during this type of refactoring, followed by details of the roles that inheritance and comment lines play in carrying out a refactoring, based around constructors, in Section 6.6. A discussion of the issues arising from this research is given in Section 6.7 and, finally, summary and conclusions are presented in Section 6.8.

6.2 Motivation and Related Work

The motivation for the study in this chapter stems from a number of issues. Firstly, there has been a large amount of interest in the criteria for carrying out refactoring. In other words, the decision as to when certain types of refactoring should be undertaken. Yet very little empirical data exists to support the question of how widespread refactoring is in practice. The results herein support earlier findings from an empirical study of a set of library classes (Counsell et al. 2003). Therein the 'substitute algorithm' refactoring (Fowler 2000) (i.e., modification of the body of a method to improve the way it functions) was found to be the most popular type of change identified. The work in this chapter is partially based on an earlier study (Najjar et al. 2003), where empirical results of transforming the constructors into creation methods were described; this work was extended to an analysis of the role that inheritance plays in the refactoring process and the potential for reduction of comment lines as a result.

Several benefits were obtained from investigating the refactoring of constructors, one of which was improved encapsulation through making the catchall constructor private. It was also found during the course of this research that the Java framework (BSF) contained very little opportunity for refactoring of its constructors. The stability of frameworks in terms of

encapsulation trends is also highlighted in Chapter 4 (see Sections 4.5.2 and 4.5.3) (cf. Counsell et al. 2002), where it was shown that out of five industrial-sized systems empirically studied, only one system (a framework) conformed to sound encapsulation principles in terms of distribution of private, protected and public attributes and methods; one feature of the four remaining systems was the existence of protected attributes and methods in classes with no inheritance links. We would thus expect a framework system to require less refactoring because of its architectural stability and, where it is needed, for it to be a relatively easy task. The work in this chapter therefore reinforces earlier results in terms of system stability and evolution.

In terms of seminal refactoring literature, the Ph.D. work of Opdyke (Opdyke 1992) presents a number of refactorings which should be applied to software. This Thesis spawned a large amount of research in the subject. Johnson and Opdyke (Opdyke and Johnson 1993) describe a study in which they illustrate how to create abstract superclasses from other classes via refactoring. They decompose the operation of creating superclasses from other classes into a set of refactoring steps, and provide examples. They also discuss a technique that can automate these steps making the process of refactoring much easier. In Johnson and Opdyke (Johnson and Opdyke 1993), some common refactorings based on aggregation, including how to convert from inheritance to aggregation, and how to reorganise an aggregate/component hierarchy are reported. They also describe how to refine aggregations by moving variables and functions between aggregate and component classes, and how to move variables and functions within inheritance hierarchies. The seminal text by Fowler (Fowler 2000) describes seventy-two types of refactoring and illustrates each type with examples using the notation of UML (Rumbaugh et al. 1998). Included therein are a number of refactorings related to constructors including *pull-up constructor body* and *remove setting method*.

Recent empirical work in the refactoring area and its automation is found in Tokuda and Batory (Tokuda and Batory 2001), where fourteen thousand lines of code were transformed automatically which would otherwise have had to be carried out by hand. Moreover, the principles of refactoring are not limited to OO languages; other languages, such as Visual Basic, have also been the subject of refactoring effort (Arsenovski 2005). The problems and

pitfalls of undertaking a simple refactoring are described in (Najjar et al. 2005). The possibility that refactorings are linked in a composite form is also described in (O'Cinne'ide and Nixon 2000). Issues associated with poor architectural design and the implication this has for refactoring are discussed widely in Brown et al. (Brown et al. 1998).

Several papers have recognised constructors as a confusing factor in the definition of OO metrics. In the OO metrics community, Briand et al. (Briand et al. 1998) identified constructors as a contributing factor to the problems of measuring cohesion and in Bansiya et al. (Bansiya et al. 1999), cohesion metrics, based on class definitions, were empirically evaluated with *and* without constructors because of the effect they had on metrics values. Developing heuristics for undertaking refactorings based on system change data has also been investigated by Demeyer et al. (Demeyer et al. 2000).

Finally, recent work by Advani et al. (Advani et al. 2005) describes the results of an empirical study of the trends across multiple versions of open source Java software. A specially developed software tool extracted data related to each of fifteen refactorings from multiple versions of seven Java systems according to specific criteria. Results showed that, firstly, the large majority of refactorings identified in each system were the simpler, less complex refactorings, for example the renaming of class features. Very few refactorings related to structural change, involving an inheritance relationship, were found. Secondly, and surprisingly, no pattern in terms of refactorings across different versions of the software was found. Results thus suggest that developers do simple ‘core’ refactorings at the method and field level, but not as part of larger structural changes to the code (i.e., at the class level).

In the next section, we detail the empirical investigation of refactoring constructors in the five Java systems referred to in Chapter 3.

6.3 Empirical Investigation

Numerous metrics have been proposed for the analysis of software; more recently the focus has been on OO software (Briand et al. 2001). Coupling and cohesion in OO systems are two areas which have received a particularly large amount of empirical attention (Henderson-Sellers et al. 1996, Briand et al. 1997b, Briand et al. 1998, Harrison et al. 1998a). A number of research papers have also tried to show how, in practice, OO systems are not exhibiting the features we expected of them. The research described in (El Emam et al. 2001) is one such example, using a C++ telecommunications framework as a basis of the study.

Many studies were and are still being carried out for the purpose of exploring the OO paradigm and gaining more understanding of how developers use the OO paradigm principles in practice (Prechelt et al. 2003, Stein et al. 2004, Mair et al. 2005, Trifu and Marinescu 2005). In fact, refactoring has recently become one of the hottest topics in the SE community for its effective role in improving the quality of software systems in order to facilitate future adaptations and extensions (Counsell et al. 2003, Kerievsky 2004, Mens and Tourwe 2004, Advani et al. 2005, Counsell et al. 2006).

6.3.1 Refactoring constructors

The refactoring investigated in this chapter was proposed by Kerievsky (Kerievsky 2004), and is included in the list of refactorings found at *www.refactoring.com.* Herein, we employed two refactorings: *chain constructors* and *replace multiple constructors with creation methods*.

We note that the RMCCM refactoring first requires the constructors of a class to be chained, so the two said refactorings are linked. According to Kerievsky (Kerievsky 2004), motivation for refactoring constructors stems from the fact that constructors do not communicate developer intention efficiently or effectively. In addition, duplicated code in a constructor obscures the real intention of the constructor since it becomes more difficult to spot any differences. Mature software systems are filled with dead constructor code,

because it is easier to add another constructor to the class than to invest time finding out invocations to specific constructors. Code bloat (unnecessary addition of code) (Kerievsky 2004) is a direct result of this programming practice and poses a danger in terms of software comprehension and maintenance.

6.3.2 Chain constructor and creation methods

The following is a variation of an example taken from (Kerievsky 2004) of a chain constructor refactoring followed by the replacement of multiple constructors with creation methods. We begin with an original class containing three constructors for a *Loan* class; the constructors, emphasising different types of loans, differ in only minor ways. The refactoring requires us to, firstly, introduce a catchall constructor and then replace the redundant constructors with creation methods. The original class definition is as follows:

```
public class Loan{
    public Loan(float notional, float outstanding,
           int rating, Date expiry){
         this.notional = notional;
         this.outstanding = outstanding;
         this.rating = rating;
         this.expiry = expiry;
     }
    public Loan(float notional, float outstanding,
           int rating, Date expiry, Date maturity){
         this.notional = notional;
         this.outstanding = outstanding;
         this.rating = rating;
         this.expiry = expiry;
         this.maturity = maturity;
     }
    public Loan(CapitalStrategy strategy,
             float notional, float outstanding, int rating,
             Date expiry, Date maturity){
         this.strategy = strategy;
         this.notional = notional;
         this.outstanding = outstanding;
         this.rating = rating;
         this.expiry = expiry;
         this.maturity = maturity;
    }
}
```

The above class contains duplicated lines of code. Four lines of code are common to all constructors, namely:

```
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
```

The chain constructor refactoring requires constructors with significant levels of duplication to be amalgamated to form a single constructor. After applying the *chain constructors* refactoring to the above class and carrying out the required testing to ensure no side-effects emerge thereafter, the Loan class becomes:

```
public class Loan{
    public Loan(float notional, float outstanding,
             int rating, Date expiry){
          this(null, notional, outstanding,rating, expiry, null);
    }

    public Loan(float notional, float outstanding,
             int rating, Date expiry, Date maturity){
          this(null, notional, outstanding, rating, expiry, maturity);
    }
    public Loan(CapitalStrategy strategy, float notional,
             float outstanding, int rating, Date expiry, Date maturity){
          this.strategy = strategy;
          this.notional = notional;
          this.outstanding = outstanding;
          this.rating = rating;
          this.expiry = expiry;
          this.maturity = maturity;
    }
}
```

The last constructor is called the *catchall* constructor, since it incorporates all the parameters and assignments of the set of constructors. By definition, it will have at least as many parameters as the largest constructor, since its signature is the set of all constructor parameters and is called from within each of the other constructors. After chaining the constructors together, the next step of the refactoring is to replace the existing constructors with creation methods (non-constructor methods), which call the single catchall constructor with appropriate parameters and null parameters whenever necessary. This gives us the following class definition:

```
public class Loan{
// catchall constructor
    private Loan(CapitalStrategy strategy, float notional,
             float outstanding, int rating, Date expiry, Date maturity){
           this.strategy = strategy;
           this.notional = notional;
```

```
            this.outstanding = outstanding;
            this.rating = rating;
            this.expiry = expiry;
            this.maturity = maturity;
    }
// creation methods

    public static Loan NoStratOrMat(float notional,
             float outstanding, int rating, Date expiry){
            return new Loan(null, notional, outstanding,
                                rating, expiry, null);
    }

    public static Loan MatNoStrat(float notional,
             float outstanding, int rating, Date expiry, Date maturity){
            return new Loan(null, notional, outstanding,
                                rating, expiry, maturity);
    }
    public static Loan MatandStrat(CapitalStrategy strategy,
                 float notional, float outstanding, int rating,
                           Date expiry, Date maturity){
            return new Loan(strategy, notional, outstanding,
                                rating, expiry, maturity);
    }

}
```

The *catchall* constructor has been declared as private, thereby aiding encapsulation by only being accessible from the creation methods. We note that in the case where the Loan class has subclasses, the constructor should be declared as protected. Having carried out these steps, in theory programmers or developers would be less likely to get confused when creating an object. By reducing the duplication where possible, developers only have to use the methods specific to that object and those methods should be more easily identifiable. There are also quantifiable benefits to be obtained as a result of the process described.

In the ensuing subsection, the types of data upon which we quantifiably investigated this refactoring are presented.

6.3.3 Data collection

The following data item was collected automatically for each of the five Java systems:

1. The number of classes containing three or more constructors. We collected classes with three or more constructors (if we were to consider two constructors, one might be the ‘default’, in which case the collection of data would be almost meaningless) to

ensure that the classes still had at least two constructors. The minimum number of constructors for a refactoring of this type is three.

The following data items were collected manually for each of the five Java systems:

2. The number of common Lines Of Code (LOC) between each pair of constructors in each class. This data was collected in order to assess the potential for the removal of duplicated lines of code. The underlying supposition is that duplicated lines of code between any two constructors are largely unnecessary; they are the result of poor maintenance and should be removed with an appropriate refactoring technique.
3. The number of comment lines surrounding each of the constructors in classes where there were three or more constructors. To our knowledge, very little work has been done on the potential for eliminating comment lines as part of a refactoring. Yet, comment line bloat may be as great an impediment to effective refactoring as poorly written code leading to the code bloat phenomenon (Fowler 2000). In the presence of continuous maintenance, a comment may become inaccurate; a developer may forget to change the appropriate comment after changing the code.

6.3.4 Counting identical lines between constructors

Counting LOC is a process normally fraught with difficulty (Rosenberg 1997). However, in order to assess the benefits of the constructor refactoring, LOC would seem to be a good indicator of the inefficiencies found in constructors, particularly as constructors tend to comprise simple assignment statements whose comparison with other statements is relatively easy. However, we still need to be clear on, firstly, what a line of code is and, more importantly, what a duplicated line of code is. We consider two lines of code as common if they are syntactically identical. (We have already seen an example of a duplicated line of code in Section 6.3.2 with four duplicated lines between all constructors.) We also insist that the types of the parameters being assigned in a constructor must also be identical for two lines to be considered identical. We finally note that in the case of *if* conditions within constructors, only each line of the *if* condition is considered for a match with other constructors (i.e., not the entire 'if').

In the next section, we analyse the data extracted from the five Java systems.

6.4 Data Analysis

In this section, "**Cons**" stands for "**constructors**" and "**Max**" for the "**maximum**" value in the sample. Table 6.1 gives the breakdown of maximum, mean, and median number of constructors found for classes with three or more constructors (here 'nc' stands for not computable).

In the context of this section 'Size' stands for the number of classes with three or more constructors. For example, Swing has 1248 classes altogether (see Section 3.3.3.2) but only 73 classes with three or more constructors.

System	Size	Total No Cons	Max	Mean	Median
GraphDraw	6	19	4	3.17	3
BSF	2	7	4	3.50	nc
Libjava	9	40	11	4.44	3
Barat	48	158	7	3.29	3
Swing	73	333	9	4.56	4

Table 6.1: Number of classes with three or more constructors for the five Java systems

From Table 6.1 it follows that the Swing and the Barat systems contain the highest number of classes with three or more constructors. However, in terms of the proportion of classes of each system as a whole, Barat contains the higher percentage (19.05%) compared with Swing (6.34%). We note that the Barat system also contained the highest proportion of abstract classes (31) amongst the 252 investigated for this system. This compares with 109 for Swing, 1 for GraphDraw, 1 for BSF and 4 for Libjava, thus emphasising the high proportion of classes with three or more constructors in the Barat system. Interestingly, the BSF system (framework) contained the least proportion of such constructors. This might suggest that either as systems evolve (and get larger) more constructors are added to classes through maintenance, or that large systems have inherently large numbers of constructors.

Table 6.2 shows the frequencies for each of the constructors in classes that have three or more constructors. For example, the Swing system contains 18 classes which have four constructors and the Barat system contains 39 classes with 3 constructors. The Libjava system is the only system containing a class with 11 constructors. From this distribution, it would appear that in both Swing and Barat, the potential for refactoring is greater than the other three systems. However, we cannot claim this until the number of duplicate lines

found in the constructors has been determined, since removal of duplicated lines of code is a key motivation of this refactoring.

No. of Cons.	GraphDraw	BSF	Libjava	Barat	Swing
3	5	1	7	39	26
4	1	1	0	7	18
5	0	0	0	0	9
6	0	0	0	1	10
7	0	0	0	1	4
8	0	0	1	0	4
9	0	0	0	0	2
11	0	0	1	0	0

Table 6.2: Frequencies for each of the constructors in classes with three or more constructors

Table 6.3 gives the breakdown in terms of the number of duplicate lines of code found for each pair of constructors in each class. For example, in the Swing system, one line of common code was found between pairs of class constructors 39 times; two lines of common code were found to exist between pairs of constructors 22 times, etc.

LOC	GraphDraw	BSF	Libjava	Barat	Swing
One	1	0	1	54	39
Two	4	0	0	3	22
Three	3	0	0	7	3
Four	3	0	0	5	5
Five	1	0	0	4	0
Six	0	0	0	5	1
Seven	0	0	0	2	0
Eight	0	0	0	1	0
Ten	0	0	0	2	0
Eleven	0	0	0	1	0
Eighteen	0	0	0	1	0

Table 6.3: Frequency of duplicated lines of code in each Java system

From Table 6.3, it can also be seen that the Swing and Barat systems contain the widest spread in terms of duplicated code (as well as having the largest number of classes with three or more constructors as stated). However, it is the Barat system which provides the greatest opportunity for refactoring. The Swing system, although containing large numbers of classes with three or more constructors, tends to have relatively low numbers of duplicated LOC. In terms of refactoring theory, the Barat system in particular would seem to have evolved with code bloat to its constructors. One class, called *AUserTypeImpl* (see Appendix A), in the Barat system has 18 lines of duplicated code between two of its constructors.

In practical terms, after refactoring these constructors using the chain constructor and then the creation methods template, we would save 222 lines of code from the Barat system, 118 lines of code from the Swing system, 35 lines from GraphDraw, just a single line from Libjava and no lines from BSF. Once again, the BSF system appears to be the system where evolution (however long) has not caused deterioration in its constructors such that they need refactoring (Perry 2002). The same can be said (to a lesser extent) for the Libjava and GraphDraw systems.

In terms of the Barat and Swing systems, while this might not seem a huge reduction in lines of code, it does represent code removed from the class constructor definitions, where faults can easily be unknowingly seeded if the code is written clumsily, or is allowed to grow improperly. We also have the benefit of possible improved comprehension given by the re-naming of the constructors to creation methods and improved encapsulation by changing the declaration of the constructor from public to private. Another positive side-effect of replacing constructors with creation methods is the possible reduction in the number of comment lines in the class definition due to the removal of those constructors. While we condone the use of comments as good practice, comments associated with code bloat should be eliminated at the same time. In our empirical investigation, significant amounts of comment lines were observed in classes surrounding the constructors and we explore this issue in Section 6.6.2.

In the following section, we present the obstacles encountered in carrying out the refactoring undertaken in this study.

6.5 Obstacles to Java Refactoring

At the outset of this empirical study, it seemed that refactoring of constructors would be an easy task to accomplish and that most classes would conform to the refactoring as stated in theory (Fowler 2000). However, there were a few notable obstacles encountered during our empirical investigation that would seem to make the process of refactoring difficult. The first of these is the varying formats which constructors can take.

6.5.1 Alternative constructor formats

The first problem relates to the tendency for Java classes to call the constructor of their superclasses using *super* with the appropriate parameter list (rather than explicitly declare a constructor of their own). Table 6.4 shows the breakdown of classes in each system which contained a call to at least one super construct and the number of classes with no calls to super.

System	Size	With super	Without
GraphDraw	6	0	6
BSF	2	2	0
Libjava	9	4	5
Barat	48	44	4
Swing	73	34	39

Table 6.4: Number of classes with and without at least one super constructor

It is clear from Table 6.4 that the super feature is a frequently used construct in Java constructors (about half of the classes considered in Swing and nearly all in Barat). Within the category of classes containing a super construct, we have two cases to consider. The first case is where classes all call the same superclass constructor. In this case, it is clear that there is no problem in terms of refactoring and identifying the catchall constructor. The second case is where classes call different superclass constructors. In this case, we have to look for the constructor that has the (super) constructor call with the longest parameter list. The following example illustrates this feature and is taken from the Swing system:

```
public JFrame() {
  super();
  frameInit();
}
public JFrame(GraphicsConfiguration gc) {
  super(gc);
  frameInit();
}
public JFrame(String title) {
  super(title);
  frameInit();
}
public JFrame(String title,
    GraphicsConfiguration gc) {
  super(title, gc);
  frameInit();
}
```

From the above constructors we consider the last constructor as the catchall that can be called from the others. The following are the constructors after refactoring:

```
public class JFrame extends Frame implements
   WindowConstants, Accessible,
   RootPaneContainer
   {
     public JFrame() {
       this(null, null);
     }
     public JFrame(GraphicsConfiguration gc) {
       this(null, gc);
     }
     public JFrame(String title) {
       this(title, null);
     }
     public JFrame(String title,
          GraphicsConfiguration gc) {
       super(title, gc);
       frameInit();
     }
     …
   }
```

Another difficulty that might be encountered by developers in refactoring is the case when we have a call to a superclass constructor, which contains conditions, as the following example illustrates (again, this example is taken from the Swing system):

```
     public JWindow(Frame owner) {
       super(owner == null?
       SwingUtilities.getSharedOwnerFrame() : owner);
       windowInit();
     }
```

This last example would make the construction of a catchall constructor difficult. The mechanics of splitting the condition into two is relatively complex and may also detract from the overall comprehensibility of the class - a major motivation for the refactoring being considered in the first place. Finally, there is the case of declaring an *anonymous* class in a constructor, again making the search for a catchall constructor difficult.

The results from the five Java systems show potential for refactoring as long as caution is exercised in certain cases. Another obstacle which may inhibit the potential for refactoring is the number of interfaces in each of the five Java systems investigated.

6.5.2 Number of interfaces

As well as variations in the style of constructors, another feature of some of the Java systems analysed was the number of interface definitions (as opposed to class definitions). Interfaces have no constructors, and their functionality is implemented by classes. Table 6.5 shows the number of interfaces and classes in each of the five Java systems studied.

System	Interfaces	Classes
GraphDraw	2	50
BSF	5	60
Libjava	4	85
Barat	155	252
Swing	96	1152

Table 6.5: Number of interfaces in each of the five Java systems

Table 6.5 shows the largest proportion of interfaces to be in the Barat system, where there are nearly as many interfaces as there are classes. One suggestion for the large number of interfaces for this system may be that the nature of a compiler is such that there are methods which multiple classes are expected to implement (i.e., functionality which has a similar use across a number of classes). An example of this might be a data structure definition/manipulation interface used by a range of classes. As well as containing the most interfaces it is interesting to note that it was the Barat system which contained the widest spread of duplicated lines of code and also contained the largest number of classes with at least one occurrence of the super construct. The key point from the data in Table 6.5 is that no opportunity exists for refactoring interfaces since, by definition, interfaces have no constructors. In contrast, in the following section we describe features of the five Java systems which from a practical perspective provide for the use of constructor refactoring.

6.6 Further Practicalities

While the mechanics of the RMCCM refactoring (Kerievsky 2004) are relatively straightforward, we have demonstrated a number of reasons why in practice the refactoring of constructors is problematic. However, the features of the Java language and the way they are used can sometimes help the refactoring process. Hereafter, we focus on the role that inheritance and the distribution of comment lines play in refactoring.

A key issue with many Java refactorings is whether the class(es) in question has(ve) subclasses or not. The dependencies due to inheritance can complicate the mechanics of the refactoring of constructors; a class with few or zero subclasses can thus simplify those mechanics (Harrison et al. 2000). Equally, the presence of comment lines is useful for software maintainers. However, code bloat (Kerievsky 2004) causes comment line bloat (developers are likely to add comment lines when they add or change a constructor) and this could cause the opposite effect to the desired one. In other words, too many comment lines are not necessarily beneficial in terms of readability and comprehensibility of the code (and comments).

6.6.1 Inheritance

Table 6.6 shows the frequencies of subclasses for the classes that have three or more constructors. For example, the Swing system has 17 classes each of which has one subclass, and in total those 17 classes have 77 constructors. On the other hand, 45 classes have no subclasses. In the Barat system, there are two classes with one subclass each, and 42 classes with no subclasses.

No subclasses	GraphDraw	Total No Cons	Barat	Total No Cons	Swing	Total No Cons
0	5	16	42	135	45	207
1	1	3	2	10	17	77
2	0	0	3	9	3	14
3	0	0	1	4	3	12
4	0	0	0	0	2	10
5	0	0	0	0	1	4
8	0	0	0	0	1	6
65	0	0	0	0	1	3

Table 6.6: Frequencies of classes with three or more constructors in three Java systems

From Table 6.6, it can be seen that both Barat and Swing have the widest spread in terms of number of subclasses of a class, whereas the GraphDraw system has only one class with one subclass. The general pattern observable from Table 6.6 is that most of the candidate classes have no subclasses. In the Swing system, 61.6% of the classes fall into this category and for Barat 87.5% of classes have no subclasses. The other two systems, BSF and Libjava, have two and nine classes, respectively, with three or more constructors. None of those classes had any subclasses.

The large number of the classes without subclasses simplifies the process of refactoring according to the constructor refactoring described in this chapter, since complications associated with subclasses are eliminated. If a class has a large number of subclasses, every change made to that class is likely to cause a ripple effect on its subclass(es). Refactoring the constructors of classes towards the root of the inheritance hierarchy is thus more problematic because of the potential for higher numbers of dependent subclasses.

The conclusion we can draw is that for the five Java systems studied, there is plenty of opportunity for applying the constructor refactoring. The mechanics of the RMCCM are simplified (in terms of effort and time spent) if the class under consideration has no subclasses.

6.6.2 Comment lines

When any refactoring is undertaken, it is likely that some reduction in, or modification of, comment lines is possible. According to Fowler (Fowler 2000), large numbers of comments around code suggest that the code is bad. Fowler also suggests that comments are often superfluous after a refactoring has taken place. As part of our study, the number of comment lines associated with constructors was collected. The following hypothesis is set to test if there is a difference in the trends of comment lines among the different Java systems under consideration.

H_0: There is no variation in the number of comment lines that surrounded, or were embedded within, constructors in Java software systems.

H_A: There is variation in the number of comment lines that surrounded, or were embedded within, constructors in Java software systems.

The non-parametric Kruskal-Wallis test (Field 2006) is used for testing differences between the five Java systems. This test analyses the ranked data. Table 6.7 shows a summary of these ranked data, the median and the mean rank values of the comment lines in each system. The test statistic is a function of these ranks (see Section 3.3.7.1.5).

System (Group)	Size (N)	Median (Number of comment lines)	Mean Rank
Graphdraw (1)	6	2.50	50.3
BSF (2)	2	8.00	52.3
Libjava (3)	9	11	73.3
Barat (4)	48	0	35.4
Swing (5)	73	29	93.5

Table 6.7: The mean rank values of the comment lines for constructors in the five Java systems

Table 6.8 shows the Kruskal-Wallis statistic *H* and its associated degrees of freedom (in this case we have 5 groups, so 5 − 1 degrees of freedom (*df*)), and the significance (see Section 3.3.7.1.5). From this table we can conclude that the Java software systems under consideration significantly vary (*p*-value is 0.00 < 0.05) in terms of the number of comment lines that surrounded, or were embedded within, constructors. Therefore, we accept the alternative hypothesis. That is, developers do not tend to follow any particular strategy in writing comment lines related to constructors, and there are varying patterns in the number of comment lines for each system under consideration.

	Comment lines
H	65.20
df	4
p-value	0.00

Table 6.8: The Kruskal-Wallis test statistic for the comment lines of constructors in the five Java systems

In terms of the refactoring described in this chapter (replacing constructors with creation methods), an opportunity arises for removing the number of comment lines when a constructor is replaced by a creation method. The basis of this claim is that the new factory method should be self-explanatory and not need as many (if any) comment lines. The only comment lines which would remain would be those surrounding the catchall constructor.

Other things remaining equal, allowing comment lines for the catchall constructor and for each of the factory methods, we would expect to eliminate the majority of the 2469 comment lines found across the five Java systems. Thus the constructor refactoring employed in this empirical study would provide us with the opportunity to reduce both lines of code and comment lines.

In the next section, we discuss some of the issues raised by this study.

6.7 Discussion

For a study of this type, a number of caveats to its validity have to be considered. The first caveat is that we chose five Java systems of largely differing application domains; systems of identical application domains may have provided more relevant results. As a riposte to this criticism, we would claim that for the results described in this chapter to be generalised, we would want systems of different application domains. The second caveat is that we chose only classes with three or more constructors to consider for this type of constructor refactoring. We accept that a class with two constructors may be appropriate for refactoring effort. However, on the basis that the two constructors might include a default constructor, we feel it inappropriate to consider for refactoring any class with less than three constructors. Finally, we feel that the study is repeatable for other Java systems and for other languages, for example C++. This to some extent alleviates the caveat that this empirical study is a one-off.

The key refactoring of Java classes described in this chapter replaced all constructors with creation methods after, firstly, introducing a catchall constructor to handle all object creation calls. In Kerievsky (Kerievsky 2004), on which this refactoring is based, it was accepted that a compromise could be reached in certain circumstances where there is an overwhelming number of constructors; the developer is at liberty to be selective in their choice of which constructors should be transformed into creation methods. In such circumstances, it is also suggested that the creation methods themselves should be subject to parameterisation, i.e., amalgamating the parameter lists of certain methods where there is ample potential for this. In this sense, selective refactoring is more appropriate.

In his refactoring text, Fowler (Fowler 2000) does suggest a number of reasons why developers do not tend to refactor. One suggestion is that they do not have the time. We therefore accept that refactoring everything in sight is not always feasible or desirable. In certain circumstances we have suggested that refactoring of constructors is a more difficult process than theory suggests. This related specifically to situations where the *super* construct was called inconsistently. We are not suggesting that these constructors should be ignored; however, because they do not fit in with the template for refactoring suggested in

(Kerievsky 2004), extra time and resources should be allocated for refactoring classes with this format of constructor call if appropriate.

The overriding point that we would like to put forward is that not all refactorings are simple in practice when real systems are being analysed. Equally, the way that certain Java features are used by developers (i.e., inheritance and use of comment lines) can influence the decision as to what to refactor. We also make the point that any refactoring will take time and resources, both of which are in short supply in general. It is clear that size has an influence on the potential number of classes with three or more constructors in a system. What is not clear is whether classes with high numbers of constructors have acquired them over time, or had that many constructors from the start; this is a topic for further work.

6.8 Summary

In this chapter we have described the results of an empirical study regarding the role that constructors can play in the context of refactoring. The five Java systems that are described in Chapter 3 were used as a basis of this study. Making constructors intelligible and easy to comprehend is the key to easy maintenance of class definitions. We have shown that refactoring of constructors according to specific principles described in Section 6.3 is feasible, capable of saving lines of duplicated code and bringing possible benefits in terms of comprehension and encapsulation through use of a private constructor. It does, however, have its difficulties, not least with the use of the *super* construct, embedded conditions in signatures and the existence of interfaces all of which make identification of the refactoring elements problematic.

Equally, the role that inheritance plays in determining which of candidate classes to choose is an important consideration. We have shown that where a large number of classes have zero subclasses, a refactoring of class constructors can be made relatively easy. Finally, a by-product of removing constructors and replacing them with creation methods provides potential for removal of comment lines around those constructors. Savings in class size, in

terms of LOC, can be made by identifying comment lines which, as a result of the refactoring, have become redundant.

CHAPTER 7 Conclusions and Future Work

In this chapter we describe the achievements of the research presented in this Thesis from three aspects. In Section 7.1 we re-visit the two stated objectives in Chapter 1 and describe how these two objectives were achieved in the Thesis on a Chapter by Chapter basis. In Section 7.2 we describe achievements on a personal level from undertaking the research and, finally, we discuss related future work in Section 7.3.

7.1 Thesis Objectives Re-visited

The objectives of this Thesis stated in Chapter One were:

1. To obtain a greater understanding of C++ and Java systems from an encapsulation perspective; this is complemented by the study of how inheritance figures in those encapsulation trends for each type of system considered.
2. To assess the quantitative and qualitative benefits of applying refactorings related specifically to the concepts of encapsulation and inheritance in Java systems. In particular, to investigate those refactorings where both encapsulation and inheritance have a strong influence.

We state how we addressed each of those objectives in turn and how the structure of the Thesis assisted in knitting the research threads together.

The foundations for satisfying Objective 1 were laid in Chapter 2 where we described pertinent literature relating to encapsulation and inheritance. Chapter 3 played an important part in establishing the assumptions and caveats on which our empirical investigations in future chapters (and which we now describe) were to be based. In Chapter 4, we empirically evaluated five C++ and five Java systems; we statistically analysed the trends in the make-up of attribute types and method types and how that make-up was composed

and could be decomposed when inheritance was taken into consideration. For example, we explored the relationship between public and private attributes for classes inside and outside an inheritance hierarchy. The same study gave interesting insights into how those systems had possibly evolved, highlighted inconsistencies and nuances of the use of inheritance and its strong relationship with encapsulation.

Chapters 5 and 6 described two further empirical studies and helped to satisfy Objective 2. The first study investigated the EF refactoring (Fowler 1999) and the opportunities for applying that refactoring in Java systems. The EF is a basic refactoring that could be considered a low-level refactoring since its mechanics are relatively simple. We identified the potential for applying that refactoring as well as the many factors that may adversely affect a developer when considering the EF refactoring. EF seems inapplicable in many cases due to lack of (public) attributes; however, application of such a refactoring paves the way for the application of other refactorings.

In Chapter 6, we investigated a more complex (or high-level) refactoring, again related to encapsulation and inheritance. The RMCCM refactoring (Kerievsky 2004) demonstrated that such high-level refactorings can provide both quantitative and qualitative benefits. The RMCCM was capable of saving lines of duplicated code; it also provided the potential for removal of some comment lines around those constructors, thus bringing possible benefits in terms of comprehension and encapsulation through use of a private constructor. It is not, however, a straightforward task and just like EF presents many challenges in terms of its mechanics.

We thus feel both of the Objectives stated in Chapter 1 have been satisfied. Essential features of OO such as encapsulation and inheritance are inextricably linked in the context of refactoring and the basic building-blocks of the OO paradigm present interesting characteristics when combined in non-trivial OO systems.

The Thesis thus informs our empirical understanding of encapsulation and refactoring issues in OO software systems.

7.2 Contribution

As described in Chapter 1, the contribution of the Thesis can be seen in light of two research strands. The first relates to the fact that very few prior studies have empirically investigated the role that encapsulation plays in OO systems and the extent to which it is used or abused in those systems, yet encapsulation is a fundamental part of the OO paradigm. Equally, the interplay between encapsulation and inheritance in an empirical sense has not received a significant amount of research attention. The distribution of different data access specifiers (whether private, protected or public) in both Java and C++ languages play a key role in helping the developer and maintainer provide code visibility where appropriate, and deny class feature access visibility where inappropriate.

The Thesis made a contribution to our understanding and knowledge in this area, in particular to the anomalies that arose with systems that had undergone regular change. The findings of Chapter 4 suggested that encapsulation was often inappropriately used; for example evidence was found of declaration of attributes as public and use of protected features in C++ classes that did not inherit from any other classes. This contravenes accepted 'good' practice. In Chapter 4 we also found that when inheritance was specifically considered from an encapsulation perspective in C++ and Java, similar anomalies were encountered – namely a proliferation of publicly defined attributes and a distinct lack of protected features. These findings were both unexpected and surprising.

Finally, from a refactoring perspective, few studies have empirically explored the potential for applying refactorings in which encapsulation and inheritance play a central part. The Thesis contributed to our understanding and knowledge of the empirical opportunities, pitfalls and practicalities of undertaking refactorings with strong ties to encapsulation and inheritance and the derived qualitative and quantitative benefits and trade-offs. Chapter 5 looked at possibilities and potential for applying the Encapsulate Field (EF) refactoring which converts a field from public to private. It was found that the number of dependent classes requiring changes as a result of applying the EF refactoring prohibited the refactoring, and the large number of classes with zero attributes rendered the same refactoring almost redundant. Moreover, some of the systems analysed were found to be

more amenable to the EF refactoring than others. In Chapter 6 we examined the refactoring of class constructors and showed the potential for improved class comprehension by the creation of non-constructor methods (as a replacement for constructors) and improved encapsulation of class features through use of a private catchall constructor. However, nuances of the Java language made this refactoring problematic. In Chapter 6, we also investigated the role that inheritance played in the choice of classes to refactor as well as patterns in comment lines among the same constructors. Again, while certain benefits in terms of reduced numbers of comment lines (for example) were achieved, nuances of the Java language again made this refactoring task problematic.

7.3 Personal Achievement

There are many things which have been learnt in terms of research practice over the course of the period of research. Firstly, research is not an easy thing to do – and we can attribute that to many things. Research is usually done under time pressure, so time management is extremely important. Secondly, it is often necessary to follow leads which may or may not be fruitful in terms of immediate results. One tenet of any empirical research is that negative results are often as interesting as positive results, so although it may be disappointing when things do not emerge from empirical research as we would like, there is always a positive side.

It is also important to be consistent in the way that research is undertaken; for example, with the application of appropriate statistical techniques. It is also important that data is kept properly.

Research is also about choosing a small area in Computer Science and investigating that area thoroughly, rather than tackling a wider area inadequately. Part of the PhD process is to be aware of, and be able to critically assess, the literature in the chosen area.

Throughout the past few years, I have learnt that completing a PhD Thesis is no more than the first step on the road to scholarly research. I believe that researchers can be more

productive after they have passed this challenging yet satisfying stage. However, it is the enriching experience that is gained through being involved in scientific research that ultimately matters. Despite the enormous stress and the ups and downs of this process, the academic and personal experience is quite rewarding and fulfilling. The skills of work organisation and setting plans and deadlines are something that is part of the learning process.

7.4 Future Work

There are some open issues for future research. These can be divided into two themes, issues related to data collection and software metrics, and those related to refactoring from the perspective of SE in general.

In terms of data collection and software metrics future work, in Chapter 3 we discussed the extent to which manual data collection for Java software compares with its automatic counterpart for the same data. Future work in this particular area would be, firstly, to expand the study to other Java and C++ systems, so enabling a comparison across those two languages from a data collection perspective. Secondly, to increase the number of metrics collected as a first step to identifying those which are inappropriate for automatic collection and thus would have to be collected manually (taking into consideration the complexity of the metrics concerned). Finally, to analyse the effect that the age of a system has on its deterioration in more detail; the results herein would suggest that software evolution, i.e., the degradation of system quality (Petrenko et al. 2007), contributes towards the issues raised regarding the differences between automatic and manual data collection.

In terms of refactoring, we plan to carry out an empirical study of fifteen common refactorings, including that of EF across a larger sample of systems to refute or support the findings in Chapter 5, and to inform our understanding of the relationships between refactorings; in particular, compound refactorings of the type described in the same chapter. Another avenue of future research is to examine trends in '*inner classes'*, i.e., nested classes, which feature heavily in the systems we studied, but were not considered explicitly

in our analysis. For example, we may find explanations for the lack of classes in the middle of the inheritance hierarchy as a result (see Chapter 5, Section 5.5.2).

Another future area of research will be to carry out experiments, supported by statistical analyses, to determine if refactoring, in general, does actually make classes easier to understand. It is clear that size has an influence on the potential number of classes with three or more constructors in a system. A further piece of future research, allied to this, will be to monitor the maintenance of systems from the time they are developed to observe the changing patterns in constructor trends.

Another area of future work could be on replicating the study described in Chapter 4 on a number of framework-based C++ and Java systems in order to ascertain whether we would get the same results as our findings; that is, framework-based systems adhere more closely to OO principles compared to the remaining types of system under study. In addition, more can be done on investigating the specific area of the usage of the *friends* facility in C++ systems (particularly in library-based systems); in particular, whether it is used for the purpose of *operator overloading*, and not as a means of accessing the class features of the class they are declared in. Consequently, lessons can be used from delving into these issues; such lessons can be utilised to avoid similar problems in the future.

Correspondingly, we could also investigate the more abstract concept of package encapsulation and the trends in Java systems that such a concept induces (Mubarak et al. 2007).

Glossary of Software Engineering Terms

The terms in this glossary are widely used in the SE community. The purpose of this glossary is to make explicit what we mean by those terms so that no ambiguity can arise in the mind of the reader. It is not a comprehensive list of the terminology employed in the Thesis.

Abstraction

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

Abstract Class

An abstract class is a class that can not be instantiated. It contains both attributes and methods, and serves as a base class from which other classes can be derived.

Aggregation

An aggregation represents the concept of whole-part and comprises a composed class and a set of component classes. The composed class is often called "whole" and the component classes are often called "parts" (Pressman 2000). For example, the relationship between the class `BankStatment` and the class `Transaction` is aggregation because the `BankStatement` contains all the details of each `Transaction`.

Attributes

Attributes are the data fields that are defined within a class and exposed by the class directly to the clients or hidden to be accessed by class members and other inherited classes from the class itself.

Bad Smells

Bad smells in code are strong indicators of problems somewhere in the code that offer opportunities for refactoring. For example, a *Duplicated Code* bad smell (Fowler 2000) appears when a block of code can be detected in more than one place. This type of bad smell suggests a number of refactorings such as: *extract method* or *pull up field*.

Base Class

A base class is a class that serves as the ancestor (parent) for an inherited class, and usually means the direct ancestor (superclass). The base class at the top of the inheritance hierarchy is often referred to as the *root* class.

Class

A class is the fundamental unit of code reuse. It is a blueprint for an object and defines the collection of attributes and methods thereof.

Class Members (Class Features)

Class members are the attributes (data fields) and methods that make up a class definition.

Cohesion

In the context of the OO paradigm, cohesion is a measure of how well the components of a class work together to perform a single, precise task. Classes with high cohesion are desirable because they can be easier to understand, reuse, and modify.

Constructor

In C++ and Java, as well as in other OO languages, constructors are special methods that have no return type and have the same name as their class. The role of constructors is to initialise newly created objects.

Coupling

The term coupling is a measure of the extent to which an OO class depends on other classes to accomplish its mission. High coupling indicates strong dependencies (between classes),

and low coupling indicates weak dependencies, thus allowing more flexibility in software systems.

C++ Header File

A header file, in the C++ language, referenced to in the Thesis, is a file with **.h* extension.

Dependent Classes

A class *x* is dependent on class *y* if there is coupling between them.

Depth of Inheritance Tree

The depth of inheritance of a class is the length of the path from the class node to the root of the tree (root class).

Design Patterns

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

Encapsulation

Encapsulation is one of the seminal principles of OO, and is sometimes known as information hiding. It is the process of separating the elements of an object into visible and invisible elements. The label **public** refers to the external aspects, i.e., the attributes/methods that a class offers to the outside world (the other classes in the system). In other words, everything under the label public is accessible from any other class in the system. The label **private** refers to the internal aspects, i.e., the features of the class which should be shielded from the outside world. The keyword **protected** is also found in a typical OO language, but relates specifically to inheritance.

Fan-in Metric

Fan-in metric is the number of functions that call a particular function. A function with a high Fan-in means that many other functions use this function, (it could also mean that the

function is implementing a number of functionalities). If the specifications of a function with large Fan-in are changed then all the calling functions that use it have to be modified.

Fan-out Metric

Fan-out metric is the number of functions a function calls. Modifying a function can cause side-effects in the functions that are called by the modified function. A maintainer of this module has to understand many other functions, thus rendering software maintenance harder and time-consuming. Functions with a large Fan-out will be more expensive to maintain.

The Fan-in and Fan-out metrics are used to estimate the complexity of maintaining software.

Framework

A framework can be defined as a set of classes closely related in terms of functions and data, and which form an independent and reusable software artifact.

Inheritance

Inheritance is one of the key differences between conventional and OO systems. It lies at the heart of the OO paradigm. Inheritance is the ability to define a new class using existing classes as a basis. The new class inherits the attributes and behaviour of the classes of which it is a subclass, and can also have attributes and methods that are specific to it. In other words, inheritance is an abstraction for sharing similarities among data structures while preserving their differences.

Inner Class

In Java an inner class is a class whose definition is placed within another class definition or a method block for use in implementing an interface or to restrict access to it. Inner classes have access to their enclosing class members.

Interface

An interface is a well defined collection of members (methods) along with their signatures but not their implementations. Interfaces cannot be instantiated.

Lines of Code (LOC)

LOC is a metric that represents the count of "non-blank, non-comment lines" in the text of a program's source code. LOC count represents the software program size.

Localisation

Localisation is a characteristic of software that refers to gathering and placing related data and processes close to each other within the boundaries of a class or object; since the basic unit of an OO system is class, localization is based on objects.

Method

A method is a named block of code within a class that can accept arguments and might return a value. Methods determine what sort of functionality a class has, how it modifies its data fields, and provides the overall class behaviour.

Get Method

It is known sometimes as *getter* method or *accessor*. It is a method that is used to access the values of an instance variable to ensure that the instance variable can only be accessed but not modified.

Set Method

It is known sometimes as *mutator* method or *setter*. It is a method that is used to modify the value of an instance variable, giving the class more control on how its variables are being modified.

Multiple Inheritance

In C++ multiple inheritance can be defined as the mechanism by which a subclass inherits from more than one immediate superclass. Java does not support multiple inheritance in terms of class structure but does allow it in the case of interfaces.

Non-primitive Class Feature

A class feature can be described as non-primitive if its type is defined as a class.

Number of Children (NOC)

Number of children metric is the number of immediate subclasses subordinate to a class in the class hierarchy; NOC measures how many subclasses are going to inherit the superclass features. The greater the number of children, the greater the potential for reuse since inheritance is a form of reuse.

Operator Overloading

Operator overloading is the mechanism that allows us to pass different variable types to the same function and produce different results.

Pull-up Constructor Body (PUCB) Refactoring

PUCB refactoring (Fowler 2000) is applicable when a number of constructors are available on subclasses with mostly identical bodies. The mechanics of this refactoring consist of creating a superclass constructor and moving the common code of the subclasses constructors to the superclass constructor, and then calling the superclass constructor as the first step in the subclass constructor, with appropriate testing and compilation.

Java Packages

Packages in Java are used to organise class files. This can be done by putting all related class files in the same directory, giving the directory a name that relates to the purpose of the classes; the directory name is itself the name of the package where the class files reside.

Package Access

In Java, the default access is called *package access*, which facilitates access to all the classes defined within the same package.

Refactoring

Refactoring is a technique of making small steps of changes to a software system in order to improve the internal structure while preserving the external behaviour.

Rename Method (RM) Refactoring

RM refactoring (Fowler 2000) changes the name of a method when its name does not reveal the purpose of the method.

Ripple Effect Change

The situation in which one makes a change in order to remove some defect and this necessitates many other changes (Bilal and Black 2006).

Software Metrics

Software metrics include all types of metrics for software productes, software processes and software resources.

Size-related Metrics

Size-related metrics are the most traditional measures used to quantify software complexity. They are simple and easy to count, such as the LOC, number of attributes and/or methods, number of system classes, etc.

Substitute Algorithm (SA) Refactoring

SA refactoring (Fowler 2000) can be applied for the purpose of replacing a complex algorithm with one that is less complex.

Superclass

Superclass is the class that a class inherits from.

Superclass Constructor

In the Java language, the keyword *super* refers to the superclass that the current class has been inherited from. When the superclass constructor needs to be called the keyword *super* should be used explicitly with appropriate arguments.

Appendix A: Some Details of Specific Classes from the Five Java Systems

Table A.1 provides some details on the classes that are used in this Thesis as examples, and which chapter and section they are mentioned in.

Class Name	From System	Mentioned in Chapter (Section)	Class Details
Object	JDK	Ch. 4 (4.5.5)	Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class
UnicodeBlock	Libjava (Java)	Ch. 4 (4.5.4&4.5.5)	A family of character subsets in the Unicode specification. A character is in at most one of these blocks. It is an inner class of Character and has 85 public attributes
Character	Libjava (Java)	Ch. 4 (4.5.4)	Wrapper class for the primitive *char* data type. It has 55 public attributes and contains UnicodeBlock as an inner class
SecurityManager	Libjava (Java)	Ch. 4 (4.5.4)	SecurityManager is a class you can extend to create your own Java security policy. It has 1 protected attribute and 7 protected methods
EventAdapterImpl	BSF (Java)	Ch. 4(4.5.5)	This class is the root for other 13 classes
Graph	GraphDraw (Java)	Ch. 4 (4.5.5)	A class for representing a graph abstractly. It has 5 private attributes and 45 methods (of which 6 are private and 39 public)
EdgePropertiesDialog	GraphDraw (Java)	Ch. 5 (5.3.1)	A dialog class for changing the properties of an edge. It has 13 private attributes and 4 methods (1 private and 3 public). It has access to (uses) all the public attributes of class DPoint3
Node	GraphDraw (Java)	Ch. 5 (5.3.1)	A Node class for use in a graph, and for display. It has access to (uses) two public attributes of class DPoint3
GraphCanvas	GraphDraw (Java)	Ch. 5 (5.5.1)	A window class for editing and displaying graphs. It has 66 attributes and 63 methods
StrictMath	Libjava (Java)	Ch. 5 (5.5.1)	Helper class containing useful mathematical functions and constants. It has 112 attributes and 38 methods
DPoint3	Graphdraw (Java)	Ch. 5 (5.3.1)	A class for holding a real 3D position. It has 3 public attributes used directly by other two classes
Loan	Kerievsky's example	Ch. 6 (6.3.2)	See "Refactoring to Patterns" Kerievsky's book or http://www.industriallogic.com/xp/refactoring/chainConstructors.html
AUserTypeImpl	Barat (Java)	Ch. 6 (6.4)	It has 3 public constructors and there are 18 lines of duplicated code between two of its constructors
JFrame	Swing (Java)	Ch. 6 (6.5.1)	It has 4 constructors and each one of them has a call to the same superclass constructor
JWindow	Swing (Java)	Ch. 6 (6.5.1)	It has 5 constructors; one of them has a call to a superclass constructor which contains conditions

Table A.1: Some details of specific classes

Appendix B: Java Tool Software Source Code

```
/*************************************************************************
**Parsing the classes is implemented in the class **ClassParser using
**java.util.StreamTokenizer class where the input stream of each Java
**class is parsed into tokens.
**The data collected for each class or interface in a file included:
**Number of inner classes
**Number of implemented interfaces and/or abstract classes
**Number of extended classes
**For class methods:
**Method name
**Total number of public, protected, private, abstract, **static, native
**methods separately
**Number of primitive and non-primitive methods
**Number of class constructors
**For class attributes:
**Total number of primitive and non-primitive attributes
*************************************************************************/
import java.io.*;
import java.lang.*;
import java.util.*;

public class ClassParser {

   public ClassParser(){};
   private StreamTokenizer in;
   public  boolean Clas          = false;
   public  boolean Pub           = false;
   public  boolean Pro           = false;
   public  boolean Pri           = false;
   public  boolean Abstract      = false;
   public  boolean Native        = false;
   public  boolean Interface     = false;
   public  boolean Static        = false;
   public  boolean Extend        = false;
   public  boolean Implement     = false;
   public  boolean Primitive     = false;
   public  boolean abstractClass  = false;
   public  String  methodName    = "";
   public  String  previousString = "anything";
   public  static  ArrayList classes = new ArrayList();
   private String[] spicialKeyWords  = {"synchronized", "transient",
                                        “final"};

   private String  primitiveTypes[] = {"int", "int[]", "long", "long[]",
```

```
                                        "float", "float[]","double",
                                        "short", "short[]","double[]",
                                        "char", "char[]", "byte","byte[]",
                                        "boolean", "boolean[]", "void"};

/**********************************************************************/
//This class has all the arrays that contain all the information about
//the scanned class

    class classContent {
       public  int StaticMethods        = 0;
       public  int StaticAttributes     = 0;
       public  int Extend               = 0;
       public  int numOfImplementation  = 0;
       public  boolean BeginingOfClass  = false;
       public  boolean EndOfClass       = false;
       public  int[]   attPrim          = new int[4];
       public  int[]   attObj           = new int[4];
       public  int[]   methodStatic     = new int[1];
       public  int[]   attStatic        = new int[1];
       public  ArrayList   Extends      = new ArrayList();
       public  ArrayList   Implements   = new ArrayList();
       public  ArrayList[] methodPrim   = new ArrayList[4];
       public  ArrayList[] methodObj    = new ArrayList[4];
       public  ArrayList[] Constructors = new ArrayList[4];
       public  String       className;
       public  String       TYPE;

       private classContent(String s, String str){
                className=s;
                TYPE=str;
            }

       public void addConstructor(int i, Object element){
                if(Constructors[i]==null)
                Constructors[i]= new ArrayList();
            }

       public void addPrimMethod(int i,Object element){
                if(methodPrim[i]==null)
                    methodPrim[i]= new ArrayList();
                methodPrim[i].add (element);
            }

       public void addObjMethod( int i,Object element){
                if(methodObj[i]==null)
                          methodObj[i]= new ArrayList();
                methodObj[i].add (element);
            }

       public void addExtends(String s ){
                Extends.add(s);
            }

       public void addImplements(String s){
                Implements.add(s);
            }
      }// end of classContent class
```

```
/*********************************************************************/
/*******************ClassParser methods and constructors**************/

  private void reinitializeAccessSpecyfiers(){
        Pub =Pro =Pri= Static= Primitive= Abstract= false;
        Native=false;
    }

  private boolean checkTheClassName(String s){
        if(!classes.isEmpty()){
            for(int i=0; i < classes.size(); i++){
               String st=((classContent)
                        classes.get(i)).className;
               if(st.equals(s)) return true;
            }
         }
         return false;
    }

  public void addNewClass(String s){
        if(Interface==true){
            classes.add(new classContent(s,"interface"));
            Interface=false;
         }
        else
           if(abstractClass==true){
              classes.add(new classContent(s,"abstractClass"));
              abstractClass=false;
            }
           else classes.add(new classContent(s,"class"));
    }

  public void printElements(){
        Iterator e;
        int j,k;
        if(!classes.isEmpty())
            System.out.println("ClassNum;Type;
                        className;extends;implements;StaticAtts;
                        StaticMethods;Private Primitive Atts;Private
                        Non-Primitive Atts;Private Constructors;Private
                        Primitive Methods;Private Non-Primitive Methods;
                        Protected Primitive Atts;Protected Non- Primitive
                        Atts;Protected Constructors;Protected Primitive
                        Methods;Protected Non-Primitive Methods;Public
                        Primitive Atts;Public Non- Primitive Atts;Public
                        Constructors;Public Primitive Methods;Public Non-
                        Primitive Methods; Package Primitive Atts;Package
                        Non-Primitive Atts;Package Constructors;Package
                        Primitive Methods;Package Non-Primitive Methods;
                        numOfConstructors");

        for(int i=0; i<classes.size();i++){
            k=i+1;
            classContent cc = (classContent)  classes.get(i);
            int sum=0;
            if(cc.Constructors[0]!=null) sum = cc.Constructors[0].size
            if(cc.Constructors[1]!=null) sum = sum +
                                          cc.Constructors[1].size();
```

```
            if(cc.Constructors[2]!=null) sum = sum +
                                            cc.Constructors[2].size();
            if(cc.Constructors[3]!=null) sum = sum+
                                            cc.Constructors[3].size();
            System.out.print( k+"; "+sum+";"+cc.TYPE +
                                            ";"+cc.className+";");
            if(cc.Extends.size() >0){
                for(int l=0;l< cc.Extends.size();l++)
                    System.out.print(cc.Extends.get(l)+",");
                System.out.print(";");
               }
            else System.out.print("0;");
            if(cc.Implements.size()>0){
                for(int p=0;p< cc.Implements.size();p++)
                    System.out.print(cc.Implements.get(p)+",");
                System.out.print(";");
               }
            else System.out.print("0;");
            System.out.print(cc.StaticAttributes+ ";"+
                             cc.StaticMethods+";");
            for(j=0; j<4;j++){
                System.out.print(cc.attPrim[j]+ ";"+ cc.attObj[j]+ ";");
                if(cc.Constructors[j]==null) System.out.print("0"+";");
                 else System.out.print(cc.Constructors[j].size()+ ";");
                if(cc.methodPrim[j]==null) System.out.print("0"+";");
                else System.out.print(cc.methodPrim[j]. size()+";");
                if(cc.methodObj[j]==null) System.out.print("0"+";");
                else System.out.print(cc.methodObj[j].size()+";");
               }
            System.out.println();
           }
      }

    private boolean detectClass(){
         if(previousString.equals("class")){
             Clas=true;
             return true;
           }
         else return false;
      }

    private boolean detectInterface(){
         if(previousString.equals("interface")){
            Interface=true;
            return true;
            }
         else return false;
      }

    private boolean detectAbstractClass(String s){
         if(previousString.equals("abstract"))
               if(s.equals("class")) return true;
         return false;
      }

    private boolean detectAbstract(String s){
         return(s.equals("abstract"));
      }
```

```
private boolean detectPublic(String s){
     return(s.equals("public"));
  }

private boolean detectProtected(String s){
     return(s.equals("protected"));
  }

private boolean detectPrivate(String s){
     return("private".equals(s));
  }

private boolean detectNative(String s){
     return("native".equals(s));
  }

private boolean detectStatic(String s){
     return("static".equals(s));
  }

public boolean detectSpecialKeyWords(String s){
     int i=0;
     boolean result=false;
     for(i=0;i<specialKeyWords.length;i++){
          result = (s.equals(specialKeyWords[i]));
          if(result) return result;
        }
     return result;
  }

private boolean detectPrimitiveTypes(String s){
     int i=0;
     for(i=0;i<primitiveTypes.length;i++)
          if(s.equals(primitiveTypes[i])) return true;
     return false;
  }

private void howManyClassesImplemented() throws IOException {
     classContent CC=getLastItemInClassesList();
     while(in.nextToken()!= StreamTokenizer.TT_EOF){
           if(in.ttype == '{' || (in.ttype ==
             StreamTokenizer.TT_WORD && in.sval.equals("extends"))){
             in.pushBack();
             break;
             }
           if(in.ttype == '/') eatComments();
           if(in.ttype == StreamTokenizer.TT_WORD)
           CC.addImplements(new String(in.sval));
      }//while end
    }

private void isExtended()throws IOException{
     classContent CC=getLastItemInClassesList();
     while(in.nextToken()!= StreamTokenizer.TT_EOF){
           if(in.ttype== '{' ||(in.ttype == StreamTokenizer.TT_WORD &&
                               in.sval.equals("implements"))){
               in.pushBack();
```

```
                    break;
                  }
               if(in.ttype == '/') eatComments();
               if(in.ttype==StreamTokenizer.TT_WORD)
                     CC.addExtends(new String(in.sval));
          }//while end
     }

   public void parse(String str){
        if(detectClass()|| detectInterface()){
            addNewClass(str);
            reinitializeAccessSpecyfiers();
            return;
           }
        if(detectAbstractClass(str)){
            abstractClass=true;
            return;
           }
        if(detectPublic(str)){
            Pub=true;
            return;
           }
       if(detectAbstract(str)){
            Abstract=true;
            return;
           }
       if(detectProtected(str)){
            Pro=true;
            return;
           }
       if(detectPrivate(str)){
            Pri=true;
            return;
           }
       if(detectStatic(str)){
            Static=true;
            return;
           }
       if(detectPrimitiveTypes(str)){
            Primitive =true;
            return;
           }
        if(detectNative(str)){
            Native= true;
            return;
           }
        if (detectSpicialKeyWords(str)) return;
     }

/**********************************************************************/
//This is the main method of the ClassParser class, it calls most of the
//remaining methods of the class
/**********************************************************************/

    public void scanListing(String fname) throws IOException {
        System.out.println("Starting scanListing of :"+fname);
        String s;
        in = new StreamTokenizer(new BufferedReader(
```

```
                        new FileReader(fname)));
in.ordinaryChar('/');
in.ordinaryChar('*');
in.ordinaryChar('(');
in.ordinaryChar(')');
in.ordinaryChar('{');
in.ordinaryChar('}');
in.ordinaryChar(',');
in.ordinaryChar('\"');
in.ordinaryChar('\'');
in.ordinaryChar(';');
in.ordinaryChar('=');
in.wordChars(95,95);
in.ordinaryChar('>');
in.ordinaryChar('<');
in.eolIsSignificant(true);
while(in.nextToken() != StreamTokenizer.TT_EOF){
    if(in.ttype == '/'){
         System.out.print(in.ttype);
         eatComments();
       }
    else if(in.ttype == StreamTokenizer.TT_WORD){
        s = in.sval;
        if(s.equals("import") || s.equals("package"))
                discardLine();
        else{
            parse(s);
            if(s.equals("extends")) isExtended();
            else if(s.equals("implements"))
                        howManyClassesImplemented();
                  else previousString = s;
           } //end of the internal else
        }//end of the external else if
    else if(in.ttype=='<'){
        while(in.nextToken()!= StreamTokenizer.TT_EOF
                                        && in.ttype!='>')
                continue;
             }
    else if(in.ttype == '('){
        if(checkTheClassName(previousString)){
             addNewConstructor();
                        eatBlock();
             reinitializeAccessSpecyfiers();
             continue;
           }
        else{
             methodName= previousString;
             addNewMethod();
             eatBlock();
             reinitializeAccessSpecyfiers();
             continue;
           }
       }
    else if(in.ttype == ';'|| in.ttype == '=' ||
                                in.ttype == ','){
        addNewAttributes( eatAttributeDeclaration());
        continue;
       }
```

```
            else if(detectStatic(previousString)&&in.ttype == '{'){
                      eatStaticBlock();
                      continue;
                    }
            else if(in.ttype=='{'){
                      classState('{');
                      continue;
                    }
            else if(in.ttype == '}') classState('}');
           }// end of While Loop
         }

   private classContent getLastItemInClassesList(){
        classContent CC;
        if(!classes.isEmpty())
           return CC = (classContent) classes.get(classes.size()-1);
        else return null;
     }

   public void classState(char c){
        classContent CC;
        if(!classes.isEmpty()){
            if(c=='{'){
               CC = getLastItemInClassesList();
               CC.BeginingOfClass=true;
               CC.EndOfClass=false;
              }
            else if (c=='}')
                for(int i =classes.size()-1; i>=0;i--){
                        CC= (classContent) classes.get(i);
                        if(CC.EndOfClass==false){
                             CC.BeginingOfClass=false;
                             CC.EndOfClass=true;
                             return;
                          }
                    }
         }
     }

   public classContent getCCClass(){
        classContent CC;
        if(!classes.isEmpty())
            for(int i= classes.size()-1;i>=0;i--){
                 CC= (classContent) classes.get(i);
                 if(CC.EndOfClass==false ) return  CC;
                }
         return null;
     }

/*********************************************************************/
//The addNewConstructor()adds number of constructors to the classContent

   void addNewConstructor(){
        classContent cc=this.getCCClass();
        if(Pri) cc.addConstructor(0,new String(cc.className));
        if(Pro) cc.addConstructor(1,new String(cc.className));
        if(Pub) cc.addConstructor(2,new String(cc.className));
        if(!Pub && !Pro && !Pri)
```

```
                cc.addConstructor(3,new String(cc.className));
    }

/*************************************************************************/
//The addNewMethod() adds  method names to classContent

  void addNewMethod(){
       classContent cc= this.getCCClass();
       if(Static==true)  cc.StaticMethods++;
       if(Primitive){
           if(Pri)cc.addPrimMethod(0,new String (methodName));
           if(Pro)cc.addPrimMethod(1,new String (methodName));
           if(Pub)cc.addPrimMethod(2,new String (methodName));
           if(!Pub && !Pro && !Pri)
                cc.addPrimMethod(3,new String(methodName));

        }
       else{
           if(Pri) cc.addObjMethod(0,new String (methodName));
           if(Pro) cc.addObjMethod(1,new String (methodName));
           if(Pub) cc.addObjMethod(2,new String (methodName));
           if(!Pub && !Pro && !Pri)
                cc.addObjMethod(3,new String (methodName));
        }
    }

  void addNewAttributes(int i){
       classContent cc=this.getCCClass();
       if(Static==true) cc.StaticAttributes += i;
       if(Primitive){
           if(Pri) cc.attPrim[0]=cc.attPrim[0]+i;
           if(Pro) cc.attPrim[1]=cc.attPrim[1]+i;
           if(Pub) cc.attPrim[2]=cc.attPrim[2]+i;
           if(!Pub && !Pro && !Pri) cc.attPrim[3]=cc.attPrim[3]+i;
         }
       else{
          if(Pri) cc.attObj[0]=cc.attObj[0]+i;
           if(Pro) cc.attObj[1]=cc.attObj[1]+i;
           if(Pub) cc.attObj[2]=cc.attObj[2]+i;
           if(!Pub && !Pro && !Pri) cc.attObj[3]=cc.attObj[3]+i;
        }
       reinitializeAccessSpecyfiers();
     }

/*************************************************************************/
//The eatAttributeDeclaration() discards an attribute declaration

  int eatAttributeDeclaration() throws IOException{
       int i=0, j=0,k=0;
       if(in.ttype==';' || in.ttype == ',') i++;
       if(in.ttype == ';')  return i;
       else
          while(in.nextToken() != StreamTokenizer.TT_EOF){
               if(in.ttype == ';') return  ++i;
               if(in.ttype =='('){
                    j=1;
                    while(in.nextToken() != StreamTokenizer.TT_EOF){
                       if(in.ttype=='(') j++ ;
```

```
                        if(in.ttype==')') k++;
                        if(j==k) {j=k=0; break;}
                       }//end internal While Loop
                  }
                 else if( in.ttype =='{'){
                        j=1;
                        while(in.nextToken() != StreamTokenizer.TT_EOF){
                              if(in.ttype =='{')j++;
                              if(in.ttype == '}') k++;
                             if(j==k) {j=k=0; break;}
                           }
                     }
                 else if(in.ttype=='<'){
                        while(in.nextToken() != StreamTokenizer.TT_EOF
                                                &&in.ttype!='>' );
                           }
                 else if(in.ttype == ',') i++;
              }
        return i;
     }

/*********************************************************************/
//The discardLine() discards a comment line of type '///'

  void discardLine() throws IOException {
       while(in.nextToken() != StreamTokenizer.TT_EOF &&
                               in.ttype != StreamTokenizer.TT_EOL);
     }

/*********************************************************************/
//The eatBlock()discards a method block with consideration to its type

  private void eatBlock()throws IOException{
       int i=0; int j=0; int k=0;
       classContent cc= this.getCCClass();
       if(Abstract==true || Native==true || cc.TYPE=="interface")
           while(true ){
                if(in.ttype == ';' || in.nextToken() ==
                        StreamTokenizer.TT_EOF) break;
              }
        else{
           in.pushBack();
           while(true) {
                if(in.nextToken() == StreamTokenizer.TT_EOF) break;
                if(in.ttype == '{') i++;
                if(in.ttype == '}') j++;
                if( i!=0 && j!=0 && i==j) break;
           }
         }
     }

/*********************************************************************/
//The eatStaticBlock() discards a static block

  private void eatStaticBlock()throws IOException{
       int i=1;
       int j=0;
       int k=0;
```

```
        while(true){
              if(in.nextToken() == StreamTokenizer.TT_EOF
                                          ||( i!=0 && i==j)){
                 reinitializeAccessSpecyfiers();
                 break;
                }
              k++;
              if(in.ttype == '{') i++;
              if(in.ttype == '}') j++;
         }
     }

/*********************************************************************/
//The eatComments() discards comment lines until hit '*/')

  private void eatComments() throws IOException {
       if(in.nextToken() != StreamTokenizer.TT_EOF){
          if(in.ttype == '/') discardLine();
          else if(in.ttype != '*') in.pushBack();
          else{
              while(true){
                   if(in.nextToken() == StreamTokenizer.TT_EOF )break;
                   if(in.ttype == '*'){
                        if(in.nextToken() != StreamTokenizer.TT_EOF &&
                                                 in.ttype =='/')
                             break;
                     }
                    else in.pushBack();
                  } // while
             } // else
        }
     }

  }
}// end of ClassParser class
```

In addition to classParser, the class 'test1', available online[1], was also used to collect the required data in this Thesis. The 'test1' class includes a number of classes: javaFilter which implements FilenameFilter, MyFile and MyDir. These classes provide the means to traverse a file tree structure selecting Java files out of the tree.

[1] http://javaboutique.internet.com/tutorials/Files_Directories/mydir.html

Appendix C: Publications

2007

1. Counsell, S., Loizou, G., Najjar, R. (2007) Quality of manual data collection in Java software: an empirical investigation. *Empirical Software Engineering: An International Journal*.

2006

2. Counsell, S., Hassoun, Y., Loizou, G., Najjar, R. (2006a) Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06)*. Rio de Janeiro, Brazil, 288-296.

3. Counsell, S., Hierons, R.M., Najjar, R., Loizou, G., Hassoun, Y. (2006b) The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. *Testing: Academic & Industrial conference - Practice And Research Techniques*. Windsor, UK, 181-192.

2005

4. Najjar, R., Loizou, G., Counsell, S. (2005). Encapsulation and the vagaries of a simple refactoring: an empirical study. *Proceedings of the International Conference on Software and Systems Engineering and their Applications (ICSSEA'05)*. Paris, France, 8 pages.

2004

5. Najjar, R., Counsell, S., Loizou, G., Hassoun, Y. (2004) The quality of automated and manual data collection processes in Java software: an empirical comparison. *CAISE'04 Workshops in connection with The 16th Conference on Advanced Information Systems Engineering*. Riga, Latvia, 101-112.

2003

6. Najjar, R., Counsell, S., Loizou, G., Mannock, K. (2003) The role of constructors in the context of refactoring object-oriented software. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*. Benevento, Italy, 111 – 120.

7. Counsell, S., Liu, X., Najjar, R., Swift, S., Tucker, A. (2003) Applying intelligent data analysis to coupling relationships in object-oriented software. *International Conference on Intelligent Data Analysis*. Berlin, Germany, 440-450.

2002

8. Counsell, S., Loizou, G., Najjar, R., Mannock, K. (2002) On the inter-relationships between encapsulation, inheritance and friends in C++ software. *Proceedings of the International Conference on Software Systems Engineering and its Applications (ICSSEA'02)*. Paris, France, 8 pages.

References

Abreu, F., Carapuca, R. (1994) Object-oriented software engineering: measurement and controlling the development process. *Revised version: Originally published in Proceedings of the 4th International Conference on Software Quality*. McLean, VA, 8 pages.

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

Arsenovski, D. (2005). "Refactoring elixir of youth for legacy VB code." Retrieved 10/02/2005, from www.codeproject.com/vb/net/Refactoring_elixir.asp.

Baker, P., Evans, D., Grabowski, J., Neukirchen, H., Zeiss, B. (2006) TRex - The refactoring and metrics tool for TTCN-3 test specifications. *Proceedings of Testing: Academic & Industrial Conference - Practice and Research Techniques*. Windsor, UK, 90-94.

Bansiya, J., Etzkorn, L., Davis, C., Li, W. (1999) A class cohesion metric for object-oriented design. *Journal of Object-Oriented Programming*, 11(8):47-52.

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

Basili, V.R., Selby, R., Hutchens, H. (1986) Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733-743.

Berard, E. (1995). "Metrics for object-oriented software engineering." Retrieved 15/02/2006, from http://www.toa.com/pub/moose.htm.

Bieman, J.M., Jain, D., Yang, H.J. (2001) OO design patterns, design structure, and program changes: an industrial case study. *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01)*. Florence, Italy, 580-590.

Bieman, J.M., Straw, G., Wang, H., Munger, P.W., Alexander, R.T. (2003) Design patterns and change proneness: an examination of five evolving systems. *Proceedings of the 9th International Software Metrics Symposium (METRICS'03)*. Sydney, Australia, 40-49.

Bieman, J.M., Zhao, J.X. (1995) Reuse through inheritance: a quantitative study of C++ software. *Proceedings of the ACM Symposium on Software Reusability*. Seattle, WA, 47-52.

Bilal, H.Z., Black, S.E. (2006) Computing ripple effect for object-oriented software. *Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*. Nantes, France, 51-61.

Bocco, M.G., Piattini, M., Calero, C. (2005) A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4(9):59-92.

Bourque, L.B., Clark, V.A. (1994) *Processing Data: The Survey Example, in Research Practice, International handbooks of Quantitative Applications in the Social Sciences*. M.S. Lewis-Beck (ed), Volume 6. Sage Publications, Thousand Oaks, CA.

Briand, L., Bunse, C., Daly, J. (2001) A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering*, 27(6):513-530.

Briand, L., Bunse, L., Daly, J., Differding, C. (1997a) An experimental comparison of the maintainability of object-oriented and structured design documents. *Empirical Software Engineering-An International Journal*, 2(3):291-312.

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

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

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

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

Briand, L.C., Wust, J. (2002) *Empirical studies of quality models in object-oriented systems*. Advances in Computers, Val. 59. Academic Press, 97-166.

Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J. (1998) *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY.

Budd, T. (2002) *An Introduction to Object-Oriented Programming*. Addison Wesley, Reading, MA.

Cartwright, M., Shepperd, M. (2000) An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786-796.

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

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

Cohen, J. (1960) A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37-46.

Cohen, J. (1968) Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit. *Psychological Bulletin*, 70(4):213-219.

Collett, D. (2003) *Modelling Binary Data*. Chapman & Hall, London, UK.

Coolican, H. (1990) *Research methods and statistics in psychology*. Hodder & Stoughton, London, UK.

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

Counsell, S., Hassoun, Y., Loizou, G., 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 (ISESE'06)*. Rio de Janeiro, Brazil, 288-296.

Counsell, S., Loizou, G., Najjar, R., Mannock, K. (2002) On the inter-relationships between encapsulation, inheritance and friends in C++ software. *Proceedings of the International Conference on Software Systems Engineering and its Applications (ICSSEA'02)*. Paris, France, 8 pages.

Counsell, S., Newson, P. (2000) Use of friends in C++ software. *Journal of Systems and Software*, 53(1):15-21.

Counsell, S., Newson, P., Mendes, E. (2000) Architectural level hypothesis testing through reverse engineering of object-oriented software. *International Workshop on Program Comprehension (IWPC 2000)*. Limerick, Ireland, 60-66.

Cronbach, L.J. (1951) Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3):297-334.

Daly, J. "Replication and a multi-method approach to empirical software engineering research." Ph.D. Thesis, University of Strathclyde, 1996.

Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M. (1996) An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. *Empirical Software Engineering-An International Journal*, 1(2):109-132.

Darcy, D., Kemerer, C., Slaughter, S., Tomayko, J. (2005) The structural complexity of software: an experimental test. *IEEE Transactions on Software Engineering*, 31(11):982-995.

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

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

DeVaus, D. (2002) *Analyzing Social Science Data*. Sage Publications, London, UK.

Eckel, B. (2000) *Thinking in Java*. Prentice Hall, Upper Saddle River, NJ.

El Emam, K., Benlarbi, S., Goel, N., Rai, S. (2001) The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630-650.

English, M., Buckley, J., Cahill, T., Lynch, K. (2005) Measuring the impact of friends on the internal attributes of software systems. *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*. Budapest, Hungary, 151-160.

Fenton, N.E., Pfleeger, S.L. (2002) *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK.

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

Fowler, M. (2000) *Refactoring: Improving the Design of Existing Code*. Addison Wesley, Boston, MA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA.

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

Hall, T., Fenton, N.E. (1997) Implementing effective software metrics programs. *IEEE Software*, 14(2):55-65.

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

Harrison, R., Counsell, S., Nithi, R. (1998a) Coupling metrics for OO design. *Proceedings of the 5th IEEE International Software Metrics Symposium (METRICS 1998)*. Bethesda, MD, 150-157.

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

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

Harrison, R., Counsell, S., Nithi, R. (1998c) An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering-An International Journal*, 3(3):255-273.

Henderson-Sellers, B., Constantine, L.L., Graham, M. (1996) Coupling and cohesion: towards a valid metrics suite for object-oriented analysis and design. *Object-Oriented Systems*, 3(3):143-158.

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

Jagdish, B., Davis, C.G. (2002) Hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4-17.

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

Johnson, R.E., Opdyke, W.F. (1993) Refactoring and aggregation. *Lecture Notes In Computer Science, Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, London, UK, 742:264 - 278.

Kanmani, S., Uthariaraj, V.R., Sankaranarayanan, V., Thambidurai, P. (2004) Investigation into the exploitation of object-oriented features. *SIGSOFT Software Engineering Notes*, 29(2):1-9.

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

Kernighan, B., Ritchie, D. (1978) *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.

Kitchenham, B.A., Hughes, R.T., Linkman, S.G. (2001) Modelling software measurement data. *IEEE Transactions on Software Engineering*, 27(9):788-804.

Kitchenham, B.A., Pfleeger, S.L. (1996) Software quality: The elusive target. *IEEE Software*, 13(1):12-21.

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

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

Kramer, J. (2007) Is abstraction the key to computing? *Communications of the ACM*, 50(4):37-42.

Laing, V., Coleman, C. (2006) Principal components of orthogonal object-oriented metrics. *Software Assurance Technology Center (White Paper Analyzing Results of NASA Object-Oriented Data)*, 323:8-14.

Laplante, P.A., Neill, C.J. (2006) *Antipatterns*. CRC Press, Boca Raton, FL.

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

Liskov, B., Snyder, A., Atkinson, R.R., Schaffert, C. (1977) Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564-576.

Lorenz, M., Kidd, I. (1994) *Object-Oriented Software Engineering Metrics*. Prentice-Hall, Englewood Cliffs, NJ.

MacDonell, S., Shepperd, M. (2003) Using prior-phase effort records for re-estimation during software projects. *Proceedings of the 9th International Software Metrics Symposium (METRICS'03)*. Sydney, Australia, 73-86.

Mair, C., Shepperd, M., Jørgensen, M. (2005) An analysis of data sets used to train and validate cost prediction systems. *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering (PROMISE'05)*. St. Louis, MI, 1-6.

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

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

Mubarak, A., Counsell, S., Hierons, R., Hassoun, Y. (2007) Package evolvability and its relationship with refactoring. *Proceedings of the International ERCIM Symposium on Software Evolution*. Paris, France, 11 pages.

Najjar, R., Counsell, S., Loizou, G. (2005). Encapsulation and the vagaries of a simple refactoring: an empirical study. *Technical Report* BBKCS-05-03-02. Birkbeck, University of London: London, UK.

Najjar, R., Counsell, S., Loizou, G., Hassoun, Y. (2004) The quality of automated and manual data collection processes in Java software: an empirical comparison. *CAISE'04 Workshops in connection with The 16th Conference on Advanced Information Systems Engineering*. Riga, Latvia, 101-112.

Najjar, R., Counsell, S., Loizou, G., Mannock, K. (2003) The role of constructors in the context of refactoring object-oriented software. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*. Benevento, Italy, 111 – 120.

O'Cinne'ide, M., Nixon, P. (2000) Composite refactorings for Java programs. *Proceedings of the Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming*. Sophia Antipolis and Cannes, France, 6 pages.

O'Keeffe, M., O'Cinne'ide, M. (2006) Search-based software maintenance. *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06)*. Los Alamitos, CA, 249-260.

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

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

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

Ostrand, T.J., Weyuker, E.J., Bell, R.M. (2004) Where the bugs are. *Proceedings of ACM International Symposium on Software Testing and Analysis*. Boston, MA, 86-96.

Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053-1058.

Parsons, D. (1994) *Object-Oriented Programming with C++*. DP Publications Ltd, London, UK.

Perry, D. (2002) Laws and principles of evolution. *Proceedings of the International Conference on Software Maintenance*. Montreal, Canada, 70-71.

Perry, D., Porter, A., Votta, L. (2000) Empirical software engineering: A roadmap. *Proceedings of the Conference on The Future of Software Engineering*. Limerick, Ireland, 345 - 355

Petrenko, M., Poshyvanyk, D., Rajlich, V., Buchta, J. (2007) Teaching software evolution in open source. *Computer*, 40(11):25-31.

Post, E. (2001) Advantages of using the object-oriented paradigm for designing and developing software. *Applied Computing, Mathematics and Statistics Group, Applied Management and Computing Division*. Lincoln, Canterbury, New Zealand.

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.

Prechelt, L., Unger, B., Tichy, W., Brossler, P., Votta, L. (2001) A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134-1144.

Pressman, R.S. (2000) *Software Engineering: A Practitioner's Approach, Sixth Edition*. McGraw Hill, Berkshire, England.

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

Rosenberg, L.H., Hyatt, L.E. (1997) Software quality metrics for object-oriented environments. *Cross Talk-The Journal of Defense Software Engineering*, 10(4):7 pages.

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

Schärli, N., Black, A.P., Ducasse, S. (2004) Object-oriented encapsulation for dynamically typed languages. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vancouver, BC, Canada, 130-149.

Schneidwind, N.F. (1992) Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410-422.

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

Shepperd, M., Ince, D. (1993) *Derivation and Validation of Software Metrics*. Clarendon Press, Oxford, UK.

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

Sheskin, D.S. (2004) *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, Boca Raton, FL.

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

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

Stein, C., Etzkorn, L., Utley, D. (2004) Computing software metrics from design documents. *Proceedings of the 42nd Annual Southeast Regional Conference*. Huntsville, AL, 146-151.

Stroustrup, B. (1991) *The C++ Programming Language*. Addison-Wesley, Reading, MA.

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

Trifu, A., Marinescu, R. (2005) Diagnosing design problems in object-oriented systems. *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*. Pittsburgh, PA, 155-164.

Viera, A.J., Garrett, J.M. (2005) Understanding interobserver agreement: the kappa statistic. *Family Medicine*, 37(5):360-363.

Weinand, A., Gamma, E., Marty, R. (1988) ET++: An object-oriented application framework in C++. *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. San Diego, CA, 46-57.

Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P. (2006) Refactoring for TTCN-3 test suites. *Proceedings of SAM'06: Fifth Workshop on System Analysis and Modeling*. University of Kaiserslautern, Germany.