A Study on Software Testability and the Quality of Testing In Object-Oriented Systems

By

Amjed Abdalhamed Abbas Tahir

A THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE

UNIVERSITY OF OTAGO

This thesis is dedicated to

the loving memory of my father, Abdalhmed, who taught me to always believe in myself.

Gone but not forgotten!

my mother, Amnah, for her constant love and support.

Abstract

Software testing is known to be important to the delivery of high-quality systems, but it is also challenging, expensive and time-consuming. This has motivated academic and industrial researchers to seek ways to improve the testability of software. Software testability is the ease with which a software artefact can be effectively tested.

The first step towards building testable software components is to understand the factors – of software processes, products and people – that are related to and can influence software testability. In particular, the goal of this thesis is to provide researchers and practitioners with a comprehensive understanding of design and source code factors that can affect the testability of a class in object oriented systems. This thesis considers three different views on software testability that address three related aspects: 1) the distribution of unit tests in relation to the dynamic coupling and centrality of software production classes, 2) the relationship between dynamic (i.e., runtime) software properties and class testability, and 3) the relationship between code smells, test smells and the factors related to smells distribution. The thesis utilises a combination of source code analysis techniques (both static and dynamic), software metrics, software visualisation techniques and graph-based metrics (from complex networks theory) to address its goals and objectives.

A systematic mapping study was first conducted to thoroughly investigate the body of research on dynamic software metrics and to identify issues associated with their selection, design and implementation. This mapping study identified, evaluated and classified 62 research works based on a pre-tested protocol and a set of classification criteria. Based on the findings of this study, a number of dynamic metrics were selected and used in the experiments that were then conducted.

The thesis demonstrates that by using a combination of visualisation, dynamic analysis, static analysis and graph-based metrics it is feasible to identify central classes and to diagrammatically depict testing coverage information. Experimental results show that, even in projects with high test coverage, some classes appear to be left without any direct unit testing, even though they play a central role during a typical execution profile. It is contended that the proposed visualisation techniques could be particularly helpful when developers need to maintain and reengineer existing test suites.

Another important finding of this thesis is that frequently executed and tightly coupled classes are correlated with the testability of the class – such classes require larger unit tests and more test cases. This information could inform estimates of the effort required to test classes when developing new unit tests or when maintaining and refactoring existing tests.

An additional key finding of this thesis is that test and code smells, in general, can have a negative impact on class testability. Increasing levels of size and complexity in code are associated with the increased presence of test smells. In addition, production classes that contain smells generally require larger unit tests, and are also likely to be associated with test smells in their associated unit tests. There are some particular smells that are more significantly associated with class testability than other smells. Furthermore, some particular code smells can be seen as a sign for the presence of test smells, as some test and code smells are found to co-occur in the test and production code. These results suggest that code smells, and specifically certain types of smells, as well as measures of size and complexity, can be used to provide a more comprehensive indication of smells likely to emerge in test code produced subsequently (or vice versa in a test-first context). Such findings should contribute positively to the work of testers and maintainers when writing unit tests and when refactoring and maintaining existing tests.

Acknowledgements

I would like to express my gratitude to a number of people who supported me throughout my PhD journey. Firstly, I would like to express my sincere gratitude to my primary supervisor Prof Stephen MacDonell for his guidance and support throughout the course of my PhD (and other related) research. Steve has been a constant source of motivation during every stage of my research. I would like also to thank my secondary supervisor Prof Michael Winikoff for his advice, guidance and for reading drafts and providing feedback during later stages of this thesis.

I am extremely grateful to my mother, Amnah, and all my siblings (Adil, Adnan, Adlah, Abbas, and Ammar) and their young families for their encouragement and unlimited support (especially spiritually) throughout my PhD journey and my life in general. I could not have done it without your support, so thank you all! I would like to especially thank Abbas for his inspiration and for the many technical discussions which have improved the outcomes of this thesis.

I thank my fellow colleagues and lab-mates at the University of Otago (and previously at AUT): Sherlock Licorish, Michael Bosu and Aftab Mughal. They have certainly helped me in making this PhD experience an enjoyable one. I also thank all fellow PhD students at the Department of Information Science for the lively discussions we had whenever we get together. In particular, I would like to thank M. Alghamdi, A. Alshaer, M. Farhangian and H. Lin.

I would like also to thank my fellow colleagues and members of the Software Engineering Research Laboratory at Auckland University of Technology, whom I shared space with during my time at AUT: Frederik Schmidt, Bilal Raza, Nadia Kasto, Minjuan Tong, Waqar Husin, Da Zhang and Dr Jacqui Whalley. I would like to especially thank Jim Buchan for his feedback and collaboration during various stages of this PhD journey.

I am grateful for the financial support I received from University of Otago (Otago's Business School Dean Scholarship) and previously from AUT (SERL special PhD Scholarship).

Last but not least, I would like to thank all my friends (around the world!) for their moral help and support, and for their understanding during my PhD journey. I am lucky to have such friends! Thank you all…

Amjed Tahir

July 2015

Table of Contents

List of Tables

List of Figures

Acronyms

Acronym	Full Description
AOP	**Aspect Oriented Programming**
CBO	**Coupling Between Objects**
CC	**Cyclomatic Complexity**
DCBO	**Dynamic Coupling Between Objects**
EC	**Export Coupling**
EF	**Execution Frequency**
IC	**Import Coupling**
IDE	**Integrated Development Environment**
LOC	**Lines of Code**
NOC	**Number of *production* Classes**
NOM	**Number of Methods**
NTC	**Number of Test Cases**
OOP	**Object Oriented Programming**
OSS	**Open Source Software**
SNA	**Social Network Analysis**
TDD	**Test Driven Development**
TLOC	**Test Lines Of Code**

Chapter 1 Introduction

1.1. Background and Motivation

Although it has been defined in a number of ways, *software testability* is commonly interpreted as being indicative of the ease with which a software artefact can be effectively tested. In the last two decades the need to achieve improvements in software testability, alongside other quality and productivity improvement goals, has become an increasingly important aim of those involved in software development. There are numerous reasons for this increased attention. It is generally acknowledged that software systems are growing larger and are becoming more complex (Sommerville et al., 2012), and yet there is some evidence that the resources directed towards testing have not been keeping pace (Mouchawrab et al., 2005). Software testing activities can be costly, requiring significant time and effort in both planning and execution, and yet they are often unpredictable in terms of their effectiveness (Bertolino, 2007). As a result some estimates suggest that testing can consume as much as 50% of the total time and cost needed for software development (Brooks, 1975, Myers et al., 2011).

Although such figures are typically associated with waterfall-like processes where testing is treated as a 'phase', the centrality of testing is not just a phenomenon of plan-based development approaches: Agile software development methods such as eXtreme Programming (XP) and Scrum also give testing significant attention in light of its importance. The practice of Test-Driven Development (TDD), for example, requires that extensive test code be developed and maintained to ensure that the 'furthermost' components of the production code work correctly (Beck, 2002). Such methods follow a test-first approach, which requires the designing and writing of test code (usually unit tests – see Chapter 2) before the production code is written. In these methods, in fact, unit tests are viewed as core, integral parts of the program (Cheon and Leavens, 2002). In further noting the importance of testing in agile approaches, Beck (1994) recommended that developers spend between 25% and 50% of their time writing tests. In short, and irrespective of the development method adopted, testing is recognised as an important but high-cost activity, and so efforts to reduce the work effort required in testing, or to improve its cost-effectiveness, are sought after in research and practice.

Initiatives designed to make software easier to test are thus directed towards improving its *testability*. However, defining and measuring testability brings significant challenges in its own right. Like many non-functional properties of software, testability has been acknowledged as an elusive concept, and its measurement and evaluation have been considered to be inherently difficult (Mouchawrab et al., 2005). Although several standards and individual studies have defined testability, they have done so in various ways, reflecting the fact that they were motivated by different purposes. These different views of software testability have directed researchers to investigate various factors of software processes, products and people that can impact – directly or indirectly – software testability. Among others aspects, testability has been previously evaluated in terms of effort expended (ISO, 2001, Freedman, 1991, Traon and Robach, 1995), test coverage (Bache and Mullerburg, 1990), and the size and quality of the test suite (Binder, 1994, Bruntink and van Deursen, 2006) (as discussed in more detail in the following chapter).

The main goals and objectives of the research reported in this thesis are discussed next.

1.2. Research Goal and Objectives

The first step towards building more testable software components is to understand the factors that have an impact on their testability. The goal of this research is to provide researchers and practitioners with a comprehensive understanding of design and source code factors that can affect the testability of a *class*. In light of their widespread use this thesis focuses on the testability of Object Oriented (OO) systems, and testability is addressed from a unit testing point of view (rather than, for example, a system testing perspective, which has been the focus of several prior studies (as described in Chapters 2 and 4)).

As noted above, testability in the literature has been addressed from a range of perspectives, depending on the way it has been defined. In this thesis we consider three different views of software testability that address three different, yet related, aspects.

1. The distribution and coverage of unit tests in relation to the *centrality*[1] of software production classes. This research utilises a combination of dynamic metrics and analysis, software visualisation (i.e., network graphs) and graph metrics to understand the coverage and distribution of unit tests in programs.
2. The impact of design and implementation factors in software production code on class testability. Previous research (such as Bruntink and van Deursen (2006), Mouchawrab et al. (2005) and Zhou et al. (2012)) has addressed the relationships between several *static* software properties and class testability. This thesis additionally investigates the relationship between two *dynamic* (i.e., runtime) software properties and class testability.
3. The relationships between design flaws (i.e., code smells) and code attributes in production code and test code. We also study the quality and design factors of unit tests (i.e., test smells) and their relationships with testability.

[1] In graph theory, *centrality* is the measure of the importance of nodes within a graph.

All three aspects are investigated through experimentation in Open Source Software (OSS) contexts, as we use data obtained from various open source projects (as described and justified in Chapter 3). An overview of the aspects of software testability addressed in this research is shown in Figure 1, and a fuller explanation of the background to this research is provided in Chapter 2. As the thesis considers three different views of software testability, that are addressed by three different experiments, the specific motivation and research questions related to each experiment (underpinned by a systematic mapping presented in Chapter 4) are set out within the chapter that contains each respective experiment (Chapters 5-7).

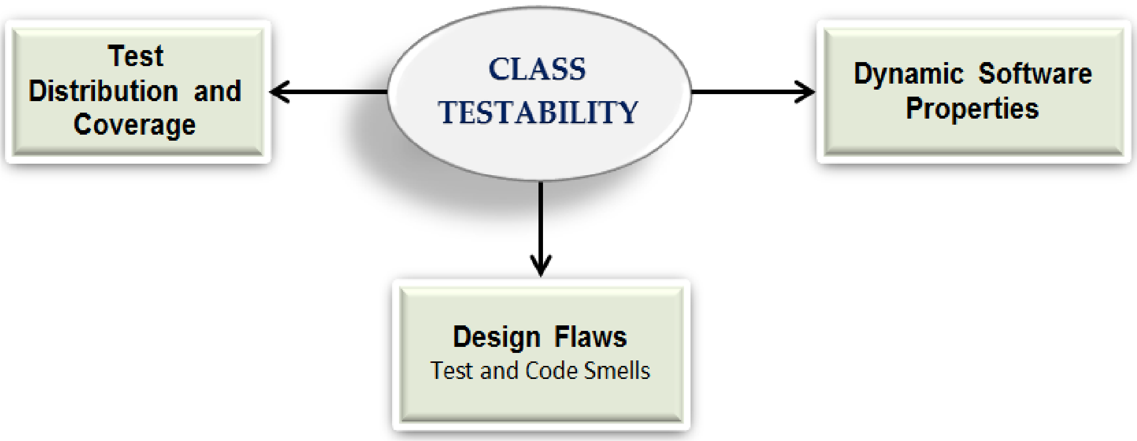


Figure 1. An overview of the testability factors addressed in this thesis.

1.3. Main Outcomes

There are four major outcomes of this work:

1. A detailed systematic mapping study of the research on dynamic software metrics and their application and usage in software quality, including dynamic metrics that are used to measure software testability. This is presented in Chapter 4.
2. A visualisation approach that combines dynamic and static information to explore unit tests' distribution. This is presented in Chapter 5.

3. A comprehensive empirical investigation of the relationship between dynamic software properties and class-level testability. This investigation is presented in Chapter 6
4. A comprehensive empirical investigation of the quality of unit tests, and the relationship between several software characteristics and the presence of design flaws (smells) in unit tests. The outcomes of this study are presented in Chapter 7.

1.4. List of Publications

During the course of this doctoral thesis, several research papers have been submitted and published in relevant research venues. At the time of submission of this work, four papers have been published in four different peer-reviewed research venues, as follows:

1. Tahir, A. & MacDonell, S. G. 2012, A Systematic Mapping Study on Dynamic Metrics and Software Quality. *IEEE 28th International Conference on Software Maintenance (ICSM).* pp. 326-335, Riva del Garda, Italy. IEEE Computer Society.

2. Tahir, A., MacDonell, S. G. & Buchan, J. 2014, Understanding Class-Level Testability through Dynamic Analysis. *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE).* pp. 38-47, Lisbon, Portugal.

3. Tahir, A., MacDonell, S. G. & Buchan, J. 2015, A Study of the Relationship Between Class Testability and Runtime Properties. *Communications in Computer and Information - ENASE 2014 Extended Versions of Selected Papers, Volume 551, pp. 63-78,* Springer .

4. Tahir, A. & MacDonell, S. G. 2015, Combining Dynamic Analysis and Visualization to Explore the Distribution of Unit Test Suites. *6th ICSE Workshop in Emerging Trends on Software Metrics (WETSoM),* Florence Italy. IEEE Computer Society

Note that there is overlap between the material contained in this thesis and the published papers.

- The first publication presents the results of a mapping study that investigates the use of dynamic metrics in software as used to measure software quality characteristics. The main aim of this work is to systematically investigate the body of research on dynamic software metrics to identify issues associated with their selection, design and implementation. The findings of this work motivated the focus on software testability, as one of the factors that can potentially benefit from the use of dynamic software metrics. This work is presented in Chapter 4 of this thesis.

- The second and the third publications[2] examine the relationships between dynamic software properties, represented by dynamic coupling (motivated by the findings of the mapping study published in the first publication) and the newly defined concept of key classes, and software testability. The results of this work are provided in Chapter 6.
- The fourth publication presents a new approach that visualises data obtained from both static and dynamic analysis to explore the distribution of unit tests in software projects. This work builds on the findings from the earlier publications. The results of this study are presented in Chapter 5 of this thesis.

1.5. Structure of the Thesis

An overview of the novel research presented in this thesis is shown in Figure 2. The remainder of this thesis is structured as follows: Chapter 2 presents a general literature review of the general topics addressed in this thesis. Chapter 3 discusses in detail the research methodology employed. Chapter 4 presents a systematic mapping study on the use of dynamic software metrics in software quality. In the first experiment presented in Chapter 5, we examine the combined use of dynamic analysis and visualisation to explore the distribution of unit tests in software systems. We then investigate the relationship between dynamic software

[2] The third publication is an extended and revised version of the second publication, which was selected to be published as a chapter in a specially edited book.

characteristics and class testability, presented in Chapter 6. We subsequently provide a detailed study in Chapter 7 on test smells and the relationship between test smells, code smells and the factors that might impact smell distribution. Finally, Chapter 8 presents a conclusion based on the findings of this thesis, it provides a discussion of the limitations of the research conducted, and it suggests a number of future research endeavours that could be carried out on topics related to those addressed here.

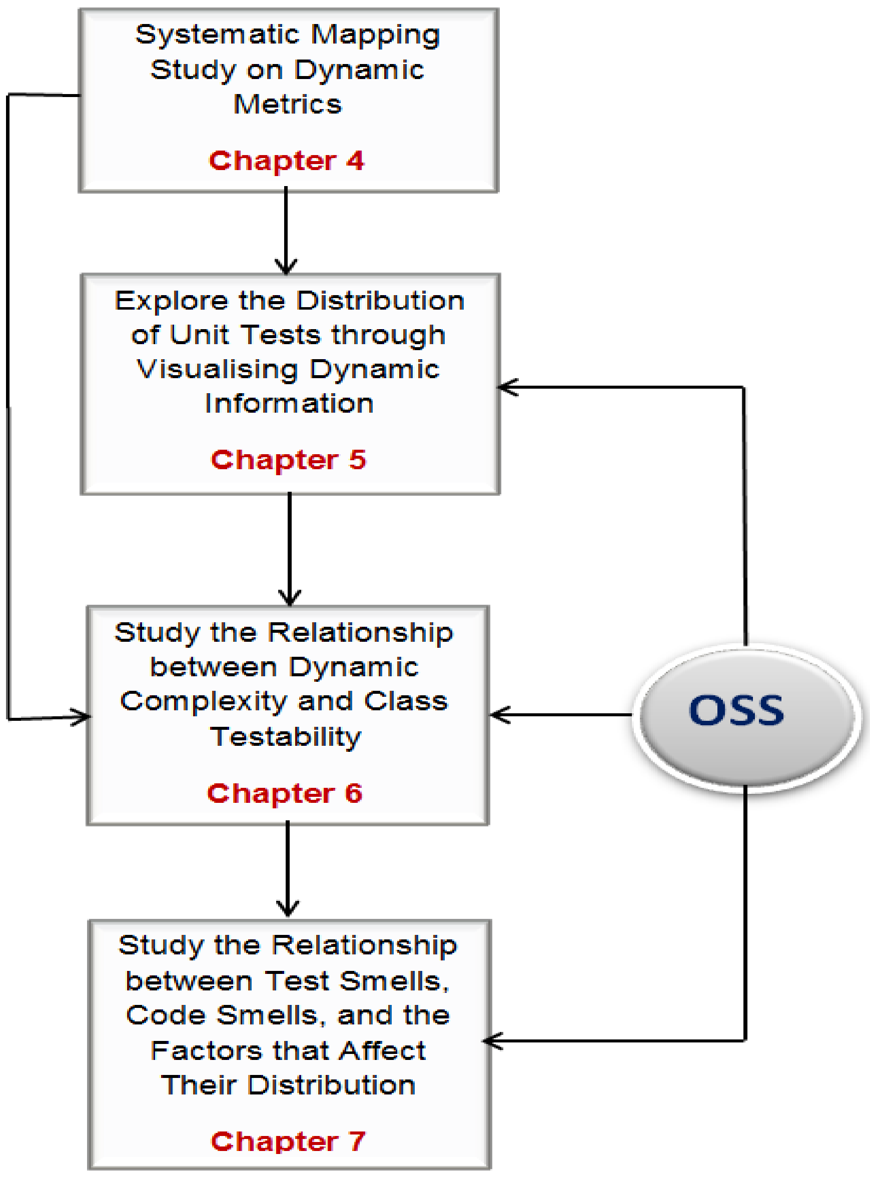

Systematic Mapping Study on Dynamic Metrics
Chapter 4
Explore the Distribution of Unit Tests through Visualising Dynamic Information
Chapter 5
Study the Relationship between Dynamic Complexity and Class Testability
Chapter 6
OSS
Study the Relationship between Test Smells, Code Smells, and the Factors that Affect Their Distribution
Chapter 7

Figure 2. **An overview of the research conducted in the thesis**

Chapter 2 Literature Review

2.1 Introduction

Although it can be defined and operationalised in many ways, the general notion of 'quality' rightly continues to attract extensive attention in the software industry, from the perspectives of both research and practice. Quality has long been viewed as an important success criterion in the software development industry (Osterweil, 1996) and it continues to play a prominent role in today's competitive software market (Elberzhager et al., 2012). Users expect high-quality software products[3] and they would normally evaluate a system's performance *as delivered*, rather than in relation to the underlying development process (McManus and Wood-Harper, 2007). As such, the quality of the final product is paramount to users – if its performance (both functional and non-functional) does not meet their expectations then they are unlikely to use the product. That said, *continued* use is very likely to be affected by the development process, as this process will have affected the internal quality of the product and will also affect the ease with which the product can be changed. It is known that adherence to quality concepts in the development process enables projects to better meet customers' requirements and expectations, and also increases the efficacy and the quality of the resulting software product or service (Leung et al., 2007). Management priorities for software development must therefore take account of development cost, time, and effort, *as well as* the quality of the development process *and* of the final intended product. Certainly the short release cycles and competitive market pressures now

[3] We use the single term *software product* to represent the collective of software products, components, systems, services, applications and apps.

predominant in the software sector are squeezing resources, and such pressures are known to have an impact on quality.

In such a context, many software organisations have therefore sought effective ways to instil and monitor the ongoing quality of their software. One of the demonstrably effective ways to obtain an indication of the quality of software is to quantify some aspects of it, utilising a range of measures. This is the intent of the work undertaken in the field of software measurement and its constituent software metrics. Measurement is considered to be fundamental to software quality, drawing as it does on quality strategies in other fields (e.g., manufacturing). Software metrics traditionally defined how various attributes of software (such as size, cost, and defects) should be measured (Grady and Caswell, 1987). More generally, metrics can be used to quantify characteristics of a software project, a development process and/or a final product.

Numerous quality attributes have been identified in the literature and captured in various industry standards (Boehm et al., 1978, IEEE, 1990, ISO, 2001). One of the key quality attributes that has been highlighted in many software quality models, and that is of particular relevance to this study, is *testability*. Testability, in simple terms, reflects how easy (or difficult!) it is to test a software product. Testability is believed to impact the cost and effort that are required during software testing (Elberzhager et al., 2012). The focus of the research conducted and reported in this thesis is on software testability at the class level. We refer to this level of testability as “class-level testability” (or simply: class testability).

The remainder of this Chapter is structured as follows: we first explain the central concept of software testability in detail by looking at its definitions and its associated factors. We then examine software measurements and their two types: static and dynamic metrics. We follow this with a reflection on the importance of test comprehension. Finally, we specifically consider the notion of quality in unit tests, and we introduce the concept of *test smells*.

2.2 Software Testability

Like many of the inherently 'good' but rather amorphous characteristics considered in software quality models, testability has been acknowledged as an elusive concept, and its measurement and evaluation have thus proved to be challenging (Mouchawrab et al., 2005). Difficulties arise, in particular, due to the many potential factors that might affect testability. In spite of this, there are good reasons to further pursue its measurement and evaluation. Software products with poor testability may be less trustworthy, even after successful testing (Bertolino and Strigini, 1996). Components with poor testability are also more expensive to repair when problems are detected late in the development process; conversely, components and products with good testability can dramatically increase the quality of software, as well as reduce the cost of testing (Gao et al., 2003). Over time numerous researchers have come to relate software testability and test efficiency to the effort and cost of conducting those tests (Gao et al., 2003, Mouchawrab et al., 2005, Bache and Mullerburg, 1990).

There is a clear economic incentive to improve testability – testing can be very expensive (Bertolino and Strigini, 1996) as it can consume up to 50% of the total cost and effort in software development (Brooks, 1975, Harrold, 2000, Myers et al., 2011). In noting the importance of testing it has been recommended that developers spend between 25% and 50% of their time writing tests (Beck, 1994). Further, this is not just a function of waterfall-like processes: contemporary Agile software engineering processes, such as eXtreme Programming (XP) and Scrum, give testing significant attention (Cornelissen et al., 2007). The practice of TDD, for example, requires that extensive test code be developed and maintained to ensure that the 'furthermost' components of the production code work correctly (Beck, 2002). In these processes, in fact, unit tests[4] - components written by the

[4] In the context of Object Oriented, unit tests are also referred to as *Test Classes*

developers/tester designed to automatically test individual production classes - are viewed not as (albeit important) after-the fact add-ons but as core, integral parts of the program (Cheon and Leavens, 2002). These unit tests provide a powerful mechanism for validating existing features when in the process of developing new functionality (Cheon and Leavens, 2002). When well-designed, the use of unit tests is known to improve software quality from the early stages of development and to enable the detection of defects more effectively when compared to other verification strategies (Runeson and Andrews, 2003). Thus, the ideal ratio of test code to production code (particularly in systems implemented with test-focused methods similar to TDD) is said to be 1:1 (van Deursen et al., 2001).

In summary, the research consensus indicates that improving the testability of software has a direct, positive impact on overall quality, but that challenges in defining then measuring and assessing software testability remain. The subsections that follow explain how testability has been defined in previous research, particularly in studies conducted and reported early in the establishment of the field. Factors that are believed to strongly influence software testability, including some novel factors, are also identified and discussed.

2.2.1 Testability Definitions

Defining and measuring testability has long presented a significant challenge, and so an extensive research effort has been directed to defining the overall characteristic as well as its associated measures. As a result, several (purportedly) general standards and many more individual studies have defined testability in a variety of ways, reflecting the fact that they were conducted in parallel and/or were motivated by different purposes. This is particularly evident in the early works in the field, a selection of which we present here. The IEEE standard glossary of software engineering terminology (1990) in fact established two definitions for testability, as follows: (1) "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met", and (2) "the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met". It is clear, then, that the IEEE definitions consider software testability from a test criteria point of view. The first definition focuses on defining test criteria and checking if these criteria have been met during actual testing, whereas the second considers test coverage and how the system's requirements have been tested. The relevant ISO standard (ISO, 2001) defines testability as "attributes of software that bear on the effort needed to validate the software product". Thus, the IEEE definitions consider software testability from a test criteria point of view, whereas the ISO definition, in contrast, considers testability based on the effort needed to test a software product.

Bache and Mullerburg (1990) defined testability based on test coverage, as "the minimum number of test cases to provide total test coverage, assuming that such coverage is possible"(p. 3). The authors measured software testability statically using a flow graph model. Freedman (1991) defined testability based on effort, time, and resources required in testing a software. Their goal was to drive towards producing easily testable – and therefore lower cost – software components. Traon and Robach (1995) extended the well-established notions of analysis of hardware testability to define software testability as "The effort needed to test and repair a considered system". In their view, software testability is concerned with not just one but three different aspects, namely, test data generation, test result interpretation, and diagnosis.

Although brief, this discussion should serve to demonstrate how testability has been viewed and defined in the seminal literature. These different views of software testability have led to multiple influential factors being identified, aligned with one or more of these different views. As we consider several of these factors (or similar) in the work presented later in this thesis we now provide a brief background explanation of these factors.

2.2.2 Testability Factors

As just noted, different views have been adopted when authors have defined software testability; researchers have therefore identified multiple factors as having an impact on the testability of software. For instance, software testability is said to be affected by the extent of the required validation, by the process and tools used, and by the representation of the requirements, among other factors (Bruntink and van Deursen, 2006). Given their various foundations it is challenging to compile a complete and consistent view on *all* the potential factors that *may* affect testability, and the degree to which these factors are *present and influential* under different testing contexts. Several are considered here, intended to represent the breadth of issues of potential influence.

In one of the earliest works of relevance Freedman (1991) extended a well-established characterisation scheme used for assessing the testability of hardware components to software components (as did (Traon and Robach, 1995)). In Freedman (1991), the author defined what is called 'domain testability'-which can be defined and measured using two attributes: observability (the ease of determining if specified inputs affect the outputs) and controllability (the ease of producing a specified output from a specific input). Traon and Robach (1995) categorized testability based on three main factors, namely, complexity of the unit/component/system under test, effort required in performing the testing of these units/components/systems, and diagnosability (allowing easy location of faults).

Lo and Shi (1998) defined three factors that they claimed to substantially affect the testability of OO systems, namely, structure, communication, and inheritance factors. Jungmayr (1999) related testability to the dependency between software components: the greater the dependency, the more tests that are required to exercise their interface; hence, the higher the dependency between components, the lower their testability.

As with other quality attributes, in order to be effectively managed (i.e., planned, monitored and controlled), software testability and its constituent factors should be modelled, measured, analysed and interpreted. The following section describes the field of software measurement and metrics and explains the types of metrics than can be used to measure the characteristics of interest in this study.

2.3 Software Measurement and Metrics

Measurement is a key process in any engineering discipline, and while there remain some questions over the 'engineering-ness' of software engineering the importance of measurement is nevertheless widely acknowledged and accepted. The main goals of software measurement are to assist project managers and software engineers in making predictions to support planning, monitoring progress to enable control, and judging the performance of the software relative to goals and objectives (Stockman et al., 1990). The quantification of software characteristics is effected using *software metrics*. Software metrics are considered by many to be fundamental to software quality (McManus and Wood-Harper, 2007), and as such they have played an important and long-established role in the analysis and improvement of software quality (Basili et al., 1996). Among other things, software metrics can help personnel to ensure, throughout the software life cycle, that quality requirements are indeed being met.

As introduced previously, software metrics are commonly considered in relation to three main categories that are related to different aspects of software development: *product*, *process* and *project* (Fenton and Pfleeger, 1998, Kan, 2002). In the software engineering body of literature, quality has been most often associated with product and process metrics. In line with this thinking, Kan (2002) classified quality measurements into two classes: in-process and end-product metrics. In-process metrics help to characterise the quality of the development process, while end-product metrics reflect the relevant characteristics of the software itself (though more correctly, and importantly, they might also reflect characteristics of intermediate products, not just the 'final' version).

Within the class of (end) product metrics there are two further sub-classes that gauge different aspects of a system: static and dynamic metrics. The ISO 9126 model (ISO, 2001) reflects these two different views on software product quality:

1. Attributes and properties that can be measured without executing or running the program – these attributes characterise the internal quality of the software and are quantified in the form of *static metrics*.
2. Attributes and properties that can be measured only at run-time (software execution) – these attributes are quantified in the form of *dynamic metrics*.

As both are important in terms of obtaining a complete view of software quality (discussed further below) both groups are explained in the following sub-sections.

2.3.1 Static Metrics

Static metrics are the group of software metrics collected by measuring non-running system representations (Sommerville, 2006) and as such they capture only the static structure of a system (Cleland-Huang et al., 2001). They are typically collected using static analysis techniques, the 'traditional' means of understanding and measuring a program (or software component), enabling the interested stakeholder to explore and analyse the source code as well as any associated products or documentation (Ball, 1999, Pirzadeh et al., 2010, Cornelissen et al., 2011). Static metrics are invariant (i.e., their values do not change whether they are collected before or during program execution). Most existing quality metrics come in static form. The most notable (while perhaps not the most useful) is the Lines of Code (LOC) metric, which measures the number of physical command lines in a program/component. McCabe's well-known indicator of complexity, the *Cyclomatic Complexity* (CC) metric (McCabe, 1976), is another widely known static metric that has been used extensively in the past for assessing the complexity of software systems.

The main advantage that comes from the use of static metrics is coverage completeness: static analysis supports the prediction of behaviour resulting from multiple scenarios/paths through the software, since it is performed without actually executing the program. Despite this high coverage level, a number of limitations have been noted with the use of static metrics. Unsurprisingly, empirical studies show that static measurement and analysis is insufficient for capturing dynamic dependencies among system modules, such as those related to polymorphism, dynamic binding, and inheritance (Arisholm et al., 2004). The presence of dead code in the production code is also difficult to detect statically (especially in the presence of polymorphism) (Zaidman and Demeyer, 2008). Furthermore, static analysis may result in the generation and/or collection of a huge amount of data (Ernst, 2003) that may be difficult to understand and summarize. These (and other) disadvantages have motivated researchers to look at possible solutions that can be provided by dynamic metrics, as now discussed.

2.3.2 Dynamic Metrics

Dynamic metrics are the sub-class of software measures that are used to capture the dynamic behaviour of a software system[5] and, like their static counterparts, they have been promoted as being directly related to several software quality attributes of interest such as maintainability and reliability (Cai, 2008, Gunnalan et al., 2005, Scotto et al., 2006). As early as 1996, Basili et al. (1996) stated that traditional static software metrics may not be sufficient for characterizing, assessing, and predicting the quality of OO systems, now the dominant structural

[5] For the purpose of this thesis, we focus on specific types of dynamic metrics that are related to particular quality attributes such as maintainability, functionality and testability. Other forms of dynamic metrics, such as performance and time-related metrics, are considered to be outside the scope of this thesis.

form for software systems; hence the need for dynamic metrics. Dynamic metrics are usually computed based on data collected during program execution (i.e., at runtime) and may be obtained from the execution traces of the code (Gunnalan et al., 2005) (although in some cases simulation can be used instead of the actual execution), and as such they can directly reflect the quality attributes of that system *in operation*.

Due to their more recent consideration, dynamic metrics have received less extensive attention in the OO metrics literature compared to that afforded to static metrics (Yuying et al., 2005). There exists a large body of research on static metrics and a rapidly growing body of work on dynamic metrics; however, research on the *factors* affecting dynamic metrics, and on any relationships with their static counterparts has been limited (Mitchell and Power, 2006, Hamou-Lhadj and Lethbridge, 2010). It has been noted that developers have tended to focus more on static rather than dynamic metrics, at least partly because static metrics are much easier to compute (Dufour et al., 2003a). There are indeed several challenges associated with the collection of dynamic metrics, including code instrumentation and possibly limited availability of the source code.

As stated briefly above, the collection of dynamic metrics can be accomplished in different ways. Most common is to collect the data by obtaining trace information using dynamic analysis techniques during software execution. Another method is to simulate runtime behaviour based on executable models and interaction diagrams (such as UML and Real-time Object Oriented Modelling (ROOM) languages). The first approach provides actual figures reflecting system behaviour, as it captures the true values that accrue under execution. The disadvantage of this approach is that it is only feasible in the later stages of development. On the other hand, simulation does not require executable code and so the metrics data can be collected at an earlier stage. However, given likely changes between design and code, as well as the greater detail available in the code, this technique tends not to be as accurate and precise as its execution-based counterpart. Despite these difficulties, both techniques have been empirically examined to collect and test several dynamic metrics (Cleland-Huang et al., 2001, Dufour et al., 2003a, Gupta and Chhabra, 2011, Yacoub et al., 1999). Both static and dynamic analyses return potentially useful information concerning software artefacts (e.g., methods and classes) and their relationships (e.g., method calls) (Stroulia and Systä, 2002). Table 1 summarises the key differences between static and dynamic metrics.

Table 1. Comparison of static and dynamic metrics

Static Metrics	Dynamic Metrics
Faster and Easier to collect	Slower and more difficult to collect
Can be obtained in early stages of development (such as design-related metrics)	In most cases, are available in later stages of development
Provides wide coverage, but shallow (less precise)	Provides narrow coverage , but very deep (more precise)
Related to structural characteristics	Related to behavioural characteristics
Basic knowledge of the software is required	Advanced knowledge of the code required
Capture only invariant properties (i.e., does not support certain OO features such as dynamic binding and relationships between objects)	Suitable for collecting unique OO features (e.g. dynamic binding and relationships between objects)

This research adopts dynamic analysis during execution, given its ability to provide more accurate and precise results in comparison to the simulation method. The intent of this research is to measure programs that have already been developed, so a prerequisite to inclusion is that the source code should be available during the measurement process.

Dynamic Analysis

Dynamic analysis is the process of analysing the properties of running programs (Ball, 1999). It focuses on a real product's execution (Ball, 1999) and occasionally or periodically requires run-time data collection to support the investigation of the properties of interest (Gupta and Chhabra, 2011). Studying the dynamic behaviour of a program can dramatically improve developers' understanding of that program, by revealing characteristics that cannot be found from statically investigating the source code alone (Corbi, 1989). Data gathered from dynamic analysis are both detailed and more precise than are available from the analysis of static features (Pacione et al., 2003, Ernst, 2003, Richner and Ducasse, 1999) in terms of reflecting runtime behaviours. Another advantage of the use of dynamic metrics is the reduced level of detail that needs to be reviewed. Examining one or a few specific execution scenarios dynamically limits the scope of investigation, resulting in the provision of detailed results *but only about those specific scenarios* (Ernst, 2003, Richner and Ducasse, 1999).

The reported disadvantages of dynamic analysis lie in two different but related points: incompleteness and limited generalisation. Incomplete coverage is one of the major arguments against the use of dynamic analysis, as the gathered data can reflect only the scenario that was executed (Cornelissen et al., 2011). However, it has been argued equally strongly that incomplete coverage of software code is not necessarily a weakness (Richner and Ducasse, 1999). That is to say, to understand a program, evaluators need sufficient information to help them to form concepts about the essential software structure, not necessarily to understand its complete structure in full detail. This remains a somewhat contentious issue. The incomplete coverage 'problem' leads to the second disadvantage: limited generalisation of the results obtained. Dynamic metrics' results may not generalise to future executions, given that the gathered data pertain solely to the scenario that was executed at a given point in time (Safari-Sharifabadi and Constantinides, 2008, Cornelissen et al., 2011). There is no way of assuring that the scenario under which the program was run is representative of all possible program scenarios and executions (Ernst, 2003). One of the obvious ways to mitigate this problem is to execute a range of scenarios that together represent all major execution paths (rather than all possible paths) according to cost-benefit. The combined results, compiled across multiple scenarios and paths, should consequently provide more complete as well as more accurate results.

While the above description is sufficient for our purposes, the concepts underpinning dynamic analysis have been investigated and explained in far more detail in the literature – the interested reader is referred to the following sources for further coverage of this technique (Graham et al., 1982, Lange and Nakamura, 1997, Pauw et al., 1998, Ernst, 2003).

The two main techniques that have been used in performing dynamic program analysis are compiled code instrumentation and source code instrumentation. Compiled code instrumentation can be useful when source code is not available; however, given its richer coverage, source code instrumentation is normally preferred when it is possible. Source code instrumentation itself has two forms: manual and automatic code instrumentation. Manual instrumentation requires manual insertion of collection points, whereas automatic instrumentation can perform the insertion task under the control of a tool. Table 2 briefly describes these various dynamic analysis techniques and summarizes their advantages and disadvantages.

One of the main challenges encountered when using software measurement for assessing product/system quality is achieving the optimum combination of metrics data. Many researchers have commented on the synergies that could be gained when combining both static and dynamic information (Rothlisberger et al., 2009, Riva and Rodriguez, 2002, Ernst, 2003). Their data may complement each other; therefore they could be used alongside one another in order to build stronger evidence about the software under investigation.

Table 2. Dynamic analysis techniques

Technique		Description	Advantages	Disadvantages
Compiled code (binary) instrumentation		Perform the instrumentation automatically, under the control of a debugger	The availability of the source code is not required	· Advanced knowledge of the binary code is needed · Slows down the execution
Source Code Instrumentation	**Manual instrumentation**	Insert data collection points *manually* in several locations in the source code	Fast, as long as the source code is understandable	· Prior knowledge of the source code is required before undertaking any analysis · The source code will be modified · It affects the source code's consistency, readability and understandability (as well as size metrics such as LOC) ·
Source Code Instrumentation	**Automatic instrumentation**	· Insert data collection points *automatically* in different locations all over the program · *Example:* Aspect-Oriented Programming (AOP)	· No need for any manual instrumentation · The source code does not need to be modified · Engineers do not need to read and understand the application's source code	· Source code has to be available · It can considerably slow down the performance

Software metrics have been widely used to quantify several aspects of the software in order to help engineers in better understanding their architecture and source code. One of the important applications of software metrics is in program comprehension. The following section explains the concept of program comprehension, followed by a discussion on test comprehension and its relationship with testability.

2.4 Program Comprehension

Program comprehension (also referred to as program understanding) is a key prerequisite to software maintenance and evolution (Mayrhauser and Vans, 1995). Extensive effort in both software engineering research and practice has been directed to supporting the understanding and maintenance of software artefacts. Program comprehension is the process of obtaining knowledge about a program and trying to understand the program using the gained knowledge (Biggerstaff et al., 1993, Mayrhauser and Vans, 1995). Program comprehension is achieved when the following occurs: "A person understands a program when they are able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program" (Biggerstaff et al., 1993).

The long-held desire to achieve high levels of software reusability and lower levels of maintenance makes program understanding and comprehension even more important (Lange and Nakamura, 1997). Program comprehension research has helped researchers, engineers and quality assurance personnel to develop effective methods to understand and maintain large and complex software systems. Program comprehension is one of the key phases of any maintenance task. Software engineers usually spend a considerable amount of time, up to 60 percent of their total maintenance effort (Corbi, 1989), trying to understand software code, especially with large software systems, before making any alteration to the software system (Ko et al., 2006, Singer et al., 1997, Pirzadeh et al., 2010). Various researchers have examined the use of a range of methods and techniques to improve program understanding. Visualisation, in particular, has been used in several previous works for the purpose of supporting developer understanding of different aspects of production code (Lange and Nakamura, 1997, Jerding and Rugaber, 2000, Cornelissen et al., 2011). Many works (including

(Systä et al., 2001, Lange and Nakamura, 1997)) have proposed and assessed various methods, techniques and tools that use data obtained from both static and dynamic analysis to support program understanding.

While more conventionally associated with core application (production) code the concept of program comprehension also applies to test code. Moreover, given the centrality of testing to contemporary development methods, maximising test code quality is itself a worthy software engineering goal. Test comprehension can be defined as the process of understanding the structure and the functionality of test suites (e.g., unit tests). This involves understanding the design of the test code, the structure and design of the test cases, and the quality of its source code. This section now considers test comprehension and understanding from two different perspectives: test comprehension (i.e., test coverage and distribution) and test source code quality.

2.4.1 Test Comprehension

Test comprehension has been the focus of several works that have sought to explicate the relationship between production and test code, while other works have considered the structure of unit tests and test suites. It has been acknowledged that understanding test code and test suites can be a particular challenge due to the fact that tests are not always well-structured (Hauptmann et al., 2012).

Test visualisation, as a means of supporting developer understanding, has also been considered in several previous works. However, and to the best of our knowledge, the combination of both *dynamic analysis* and *visualisation* has not been addressed in previous research. Cornelissen et al., (2007), based on information obtained through dynamic analysis (though generated via simulation), used UML sequence diagrams to visualize test cases to gain knowledge about the structure of software in order to support program understanding. They asserted that such visualisations could be beneficial in program understanding and for documentation purposes. Visualisation of test code dependencies was used by van Rompaey and Demeyer (2008), to localise unit tests and to investigate the relationship between test and production code. Their focus was on both the composition of and dependency between test and production units as well as among the unit tests themselves. The dependency information was obtained from static analysis. The authors recommended that size and complexity information of the various software components should also be considered to provide a more detailed and comprehensive assessment of the proposed visualisation approach. In similar work, Zaidman et al. (2011) used visualisation to investigate the co-evolution between software production code and test suites. Their study focused on mining software history information from repositories in order to detect testing information from different versions of software projects. The authors observed a significant correlation between test effort (i.e., test-writing activity) and test coverage levels in different releases. The work also proposed three different visualisation views that could be used to study how test code co-evolves over time between different releases.

Hauptmann et al. (2012) used a clone detection technique (i.e., finding similar parts of the software artefacts) to identify and locate tests in order to support better understanding of those tests. The technique was applied to 4000 tests across seven industrial systems. In general, clone detection was found to provide useful information for targeting test automation effort. The findings also revealed that significant numbers of clones existed in all examined "manually written" tests.

Other works have focused on studying the correlation between different software characteristics and software testability. The work of Bruntink and van Deursen (2006) studied the relationship between several external OO metrics and class testability. The authors found a strong association between several class-level metrics, such as size and complexity metrics, and unit test size, including the number of test cases and the lines of code per unit test. Five different software systems, including one OSS, were traversed during their experiments. However, the authors did not find any evidence of correlations between inheritance and/or dependency related metrics, such as Coupling Between Objects (CBO), and the proposed testability metrics. This is likely to be because the metrics were considered in a static form. Such relationships can be confirmed through evaluation at the object level using runtime analysis. In a similar study, Badri et al. (2011) investigated the relationship between cohesion and testability using the a set of static Lack of Cohesion metrics. They found a significant relationship between this measure of static cohesion and software testability, where testability was measured using the metrics suggested by Bruntink and van Deursen (2006). In a more recent study, Zhou et al. (2012) analysed the relationship between 80 different structural static metrics (including size and complexity metrics) and class testability (as in Bruntink and van Deursen (2006), class testability is also measured in terms of unit test size). Although the study confirmed that there is a statistical relationship between static size and complexity metrics and class testability, it did not confirm if these attributes can be used to accurately predict the testability of a class.

The above section discusses the concept of test comprehension. Given the particular focus in this work on code quality, test quality and testability the following section provides an overview of the concepts of code and test smells in detail.

2.4.2 Test and Code Smells

The term *code smells* was coined to refer to parts of code that 'scream out' to be refactored (Fowler et al., 1999). Code smells reflect design flaws and/or implementation issues in the source code that are known to have a negative impact on software quality attributes such as readability, understandability and maintainability (Abbes et al., 2011, Yamashita and Counsell, 2013). Code smells therefore indicate code structures that can lead to difficulties during software evolution and maintenance. While they become evident, code smells can be the result of poor design decisions (also known as anti-patterns). Previous empirical studies have highlighted several relationships between code smells and software artefact characteristics such as size (Yamashita and Counsell, 2013), change- and fault-proneness (Khomh et al., 2012), program comprehension (Abbes et al., 2011) and other maintenance-related tasks (Sjoberg et al., 2013, Yamashita, 2014). Given their negative impact on both the software and the work of developers, code smells should be considered for treatment through suitable refactoring actions.

Researchers have also used the term *test smells* to refer specifically to smells that affect only unit tests/test classes. The term was first defined by van Deursen et al. (2001) and was further explained by Meszaros (2006). Test smells, as with code smells, can result from the poor design or implementation of a unit test.

Society's growing reliance on software has led to increased research attention being directed to the study, and prevention, of software quality issues as indicated by smells. Researchers have therefore been working to provide empirical evidence of the impact of code smells on software artefacts and processes. In contrast, there has been relatively little attention given to the study of test smells and their impact on software artefacts and activities.

A. Code Smells

In 1999, Fowler et al. (1999) introduced the notion of code smells by providing an explanation of 22 different structures that negatively affect software programs. The authors also suggested a set of refactoring techniques that could be applied to eliminate these code smells. Soon afterward a taxonomy of code smells and a set of possible relationships between different smells was proposed (Mäntylä et al., 2003). More recently, Zhang et al. (2011) reported a detailed review study of the research on code smells. The authors found that the *Duplicated Code* (or code clones) smell has received the most research attention in the literature. The authors also reported that only a few works have studied the impact of code smells on software programs, with the majority of studies instead focusing on designing and developing code smell detection techniques and tools. Examples of code smell tools are presented in Marinescu (2004), Tsantalis and Chatzigeorgiou, 2009 (2009) and Moha et al. (2010). Code smells detection techniques include both manual (such as Travassos et al. (1999)) and automated methods (such as Moha et al. (2010) and Marinescu and Ratiu (2004)). Mäntylä and Lassenius (2006) compared manual (subjective) code smell detection methods with automated detection approaches, and found that experienced developers usually report more complex smells than less experienced developers.

Several empirical studies have been directed towards studying the impact of code smells on software quality attributes. For instance, Sabane et al. (2013) studied the impact of 13 different code smells on class testability. Class testability was measured based on the number of test cases required to test individual production classes using the minimal data member usage matrix (MaDUM) technique. This technique suggests a larger number of test cases than other similar techniques. The study found that, on average, production classes with code smells required substantially more test cases compared to classes without smells. Some particular smells, such as *Blob*, *Anti-Singleton* and *Complex Class,* are shown to be more strongly associated with the number of test cases in a unit test than other smells.

D'Ambros (2010) found that an increase in the number of code smells in the code is more likely to introduce faults and bugs. However, the authors did not find any particular code smell that consistently correlates with the number of faults across all examined systems. Khomh et al., (2012) also investigated the impact of 13 code smells on change- and fault-proneness. They studied 54 releases of four different systems and found that classes with code smells were more change-prone and fault-prone than others. In a more recent study, Hall et al. (2014) investigated the impact of five under-studied code smells on faults in three large OSS. The *Switch Statement* smell was found to have no effect on faults in the systems examined.

Other smells show varied impacts on faults in different systems. For example, *Data Clump* was found to be associated with increased incidence of faults in one system, but reduced faults in two other systems. *Middle Man* and *Speculative Generality* (individually) were found to be related with reduced faults in at least one system.

Abbes (2011) conducted a controlled experiment on the impact of two code smells (i.e. *Blob* and *Spaghetti code*) on program comprehension and understanding using students and professionals as subjects. The authors found that the appearance of one code smell in the code does not significantly impact its understandability when compared to code that does not contain either of the two smells. However, the *combination* of Blob and Spaghetti Code negatively (and significantly) affected the subjects' comprehension and understanding of the code.

A series of recent studies (Yamashita, 2014, Sjoberg et al., 2013, Yamashita and Counsell, 2013, Yamashita and Moonen, 2013) intensively investigated the impact of code smells on software maintainability and maintenance tasks across multiple case studies. Yamashita and Counsell (2013) found the number of code smells at the system level to be correlated with system size. However, their study also suggested that code smells may not be sufficient for comparing systems that are significantly different in size, but can potentially be useful for comparing systems of similar size. Yamashita and Moonen (2013) found that the proportion of problems associated with code smells was not as large as they initially expected, with only 30% of maintenance problems being related to components containing code smells. Finally, Yamashita (2014) stated that the *Interface Segregation Principle* smells are more strongly associated with maintenance issues than other studied smells. Sjoberg et al. (2013) investigated, through a controlled industrial experiment, the impact of 12 code smells on maintenance effort. They found that none of the 12 investigated smells was significantly associated with increased maintenance effort. The authors go on to assert that code size and work practices that limit the number of changes in the code may be more beneficial, from a maintenance point of view, than focusing on code smells.

To summarize, previous studies have shown that code smells might be harmful in software systems from a range of perspectives. Authors have intensively studied the impact of code smells on software maintenance and evolution. In general, code smells (combined or sometimes only individual smells) have been shown to negatively impact program comprehension (Abbes et al., 2011) and testability (Sabane et al., 2013), and to increase the possibilities of developing faults and introducing changes in the associated software systems (Khomh et al., 2012).

B. Test Smells

van Deursen et al. (2001) defined and explained a set of smells that are likely to impact unit tests in object-oriented systems. To distinguish these special code smells from other general code smells the authors used the name "*test smells*" to mark those smells that affect unit tests only, and not the production classes. The authors followed the same approach that was used by Fowler et al. (1999) to define code smells, and several refactoring techniques have been suggested to overcome these smells. One example of these smells is *Assertion Roulette*, which is also known to be one of the most diffuse (i.e., common) smells in software systems (Qusef et al., 2014, Bavota et al., 2014). This smell appears when a test case comes with too many assertions. This smell affects developers' error traceability (and also maintainability), since if one of the assertions fails, it makes it hard to identify where the error has occurred. *General Fixture* is another example of a test smell that appears when a test fixture[6] is too general (i.e., the fixture is set to test multiple classes) and the class under test only accesses part of this fixture. This smell affects test comprehension and understandability, as it could be hard for developers other than those who wrote the test to trace back the main target of the

[6] in the *xUnit* framework, this is also known as test *setup*

test in the fixture. Meszaros (2006) provided a further comprehensive explanation of these smells and their possible effects.

Most of the work on test smells has focused on designing and implementing smell detection techniques and tools (mirroring the situation with respect to code smells). Van Rompaey et al. (2007) proposed a set of metrics that can be used to detect two test smells i.e., *General Fixture* and *Eager Test*. The authors later proposed a test smell detection tool for the xUnit framework called *TestQ* (Breugelmans and Van Rompaey, 2008). Greiler et al. (2013a) presented a tool called *TestHound* that targets smells related to the test fixture. The authors defined six different test fixture related smells, including the well-known *General Fixture*. Besides the identification of smells, the proposed tool can also provide recommendations for refactoring opportunities to overcome these smells. Reichhart et al. (2007) presented a tool called *TestLint* that detects, analyses and quantifies 27 different test smells in *Smalltalk* programs.

Although many works have considered the issue of test smells, to the best of our knowledge there are only two studies that have empirically investigated test smells and their impact on software artefacts. Greiler et al. (2013b) studied how test fixture smells are distributed in OSS and how they evolve over time by analysing several releases of five OSS. The main finding of their study shows that test fixture smells do not continually increase over time (from one release to another), even when the system's complexity increases. The study also provided evidence of a significant correlation between the number of tests cases and the number of test fixture smells in a system. More recently, Bavota et al. (2014) conducted two empirical studies on test smells. The first study was concerned with the distribution of test smells and investigated unit tests in 27 different Java-based systems (25 OSS and 2 industrial projects). The authors found that test smells are widely distributed in both OSS and industrial systems. Almost 86% of the unit tests analysed contained at least one test smell. In the second study, the authors conducted a controlled experiment using 61 students and practitioners to study the impact of test smells on program comprehension and understanding during maintenance tasks. The authors reported that the presence of test smells in unit tests negatively impacted developers' comprehension during maintenance activities.

The focus of most of these previous studies was on the effect of test smells on software maintenance activities such as program comprehension and understanding. Many of these previous studies, including those reported in the last year, have suggested the need for more empirical investigations of the impact of test smells on different software artefacts.

Having presented the relevant background literature, we turn now to discuss the research methodology used in this thesis. The research methodology chapter that follows includes detailed information concerning the techniques and methods employed to study the range of aspects of testability and test quality addressed in subsequent chapters of the thesis.

Chapter 3 Research Methodology

3.1 Introduction

Computer science, information systems, and software engineering are all considered as relatively new research disciplines compared with other more established fields, including many of the social and physical sciences. Moreover, these disciplines draw on a number of established foundations. As a wide-ranging discipline, software engineering, in particular, leverages pure mathematics, logic and statistics but also engineering, psychology and sociology (as reflected in the IEEE Software Engineering Body of Knowledge (SWEBOK)). As a result it can utilise a broad range of approaches to research, depending on the particular focus of the work at hand. Research that seeks to solve problems through the building and evaluation of novel artefacts – be they concepts, models, processes or tools – increasingly derives methods from the science of design.

The research reported in the chapters that follow is empirical – that is, it primarily draws on observation and data rather than on prior theory (as general theories of software practice are still nascent) – and as such it follows a now well-established pattern for empirical software engineering research. Empirical methods have been used extensively in engineering fields in general, and in the software engineering context in particular in the last twenty years. While not a 'pure' engineering discipline due to the intangible nature of the software product, software engineering is a field highly influenced by engineering schools, and engineering thought, as it works explicitly to link theory and practice (Nunamaker et al., 1990). Empirical software engineering aims to connect theory and models evident and observable in real-life software engineering problems and solutions. The general form of empirical software engineering research and practice is shown in Figure 3.

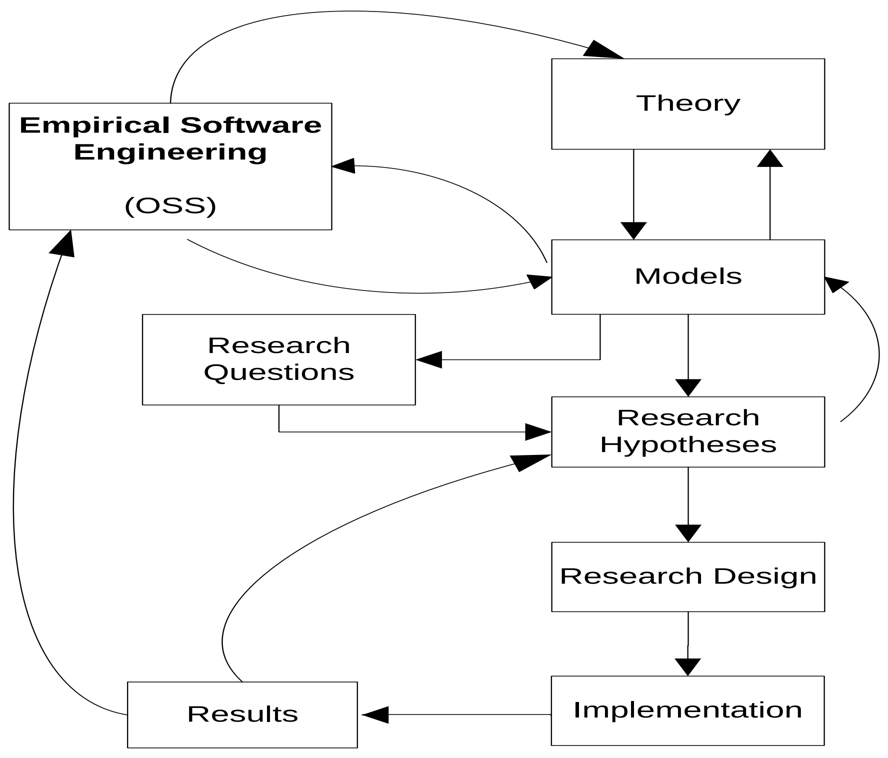


Figure 3. The empirical software engineering model used in this thesis

This chapter presents the general aspects of the research methodology used in this thesis. The more specific aspects of the methodology, including the specialist methods that are applied in each particular experiment, are presented in each relevant chapter (Chapters 5, 6 and 7). Note that Chapter 4 presents a systematic review study; as such it follows a separate method (described in that chapter) to those employed in the empirical analyses that follow it.

The remainder of this chapter is structured as follows: Section 3.2 presents the experimental design of the thesis, including a discussion on system selection and the statistical procedures used to analyse the data, Section 3.3 defines and justifies the metrics used in the thesis, Section 3.4 describes the data collection procedures and methods, Section 3.5 discuss the possible threats to the experimental validity and Section 3.6 provides a summary of the chapter.

3.2 Experimental Design

Empirical software engineering research activities may be conducted in many forms, including field studies, surveys, laboratory experiments, and case studies. A wide range of these and other research methods have therefore been employed in both academic and industrial software engineering research. The choice of plausible and appropriate methods depends on several factors, such as the theoretical stance of the researcher, access to resources, and the nature of the research questions posed by the researcher (Easterbrook et al., 2008, Wohlin et al., 2012). Given that this thesis addresses a number of research questions each of which investigates a different aspect of software testability, the specific research questions and hypotheses examined in this thesis are presented in each relevant Chapter (Chapter 5-7).

As the current research is empirical in nature, five main methods relevant to this form of software engineering research are suitable in principle: *Laboratory Experiments*, *Case Studies*, *Surveys*, *Ethnographies*, and *Action Research*. In-depth discussion of these methods and their relevance to software engineering research can be found in Easterbrook et al. (2008), Wohlin et al. (2006) and Wohlin et al. (2012). This thesis presents a number of *laboratory experiments* that examine several research questions and hypotheses.

Experimental methods are commonly used in engineering, physics and medicine research. As in other disciplines, experiments can be particularly helpful in software engineering as they enable the researcher to work with a limited scope of effects. The most widely used form of experiment in software engineering is the laboratory experiment. This type of experiment provides a means of examining an approach (or a method, technique, tool and so on) in a controlled environment. In this type of experiment, one or more independent variables are manipulated to vary their effect on one or more dependent variables (Easterbrook et al., 2008) and in software engineering these effects are typically analysed by performing appropriate statistical analyses (Wohlin et al., 2006). In our case the independent variables are not manipulated as such, as they are drawn from existing software systems, but the opportunity to consider the relationships between and effects of differences in independent variables on potentially dependent variables still applies. In this thesis, we are using secondary sources compiled by others rather than sources we have obtained directly ourselves. Specifically, these experiments enable us to statistically examine the presence and strength of any relationships between characteristics of different software artefacts and software testability. The hypotheses that reflect these relationships are laboratory tested using a number of OSS. The principal, pragmatic reason for using OSS is their ready availability. OSS are publicly available and, in many instances, their source code is completely accessible[7]. The OSS selection criteria applied in this research and the resultant list of the selected OSS are explained in more detail in the following section.

3.2.1 Open Source Systems

Much of the research reported in this thesis utilises data obtained from OSS. As the name implies, OSS grant free and full access to software projects. Nine different OSS have been selected for use in the experiments conducted in this research and reported in subsequent chapters. These OSS were identified from similar previous empirical studies as well as from well-known OSS repositories such as SourceForge[8], GitHub[9]and Google Code[10]. The systems were selected based on the selection criteria that all systems should:

- be fully[11] written in Java (Java is one of the most widely used OO programming languages in the OSS domain, based on the number of projects

[7] In some cases, parts of source code might not be publicly available.
[8] http://sourceforge.net/
[9] https://github.com/
[10] https://code.google.com/
[11] Only comprising Java code and not other languages

developed in Java,[12] and it is also considered to be one of the most popular programming languages in terms of the number of developers using it[13])

- be fully open source (i.e., giving unrestricted access to all core project artefacts)
- contain a reasonable number of unit tests (i.e., at least 20 observation points in each system, for statistical significance[14]).

A further, important consideration in system selection was applied across the *set* of systems, rather than to each individual system, and that was that the systems should be of different sizes. Consideration of systems of different sizes is intended to enable assessment of the *scalability* of the analyses conducted in this work, which can help in generalising the findings presented here. The goal is to be able to conduct experiments without having to make any assumptions regarding the size of the software system being examined. Based on the above criteria a set of nine OSS were selected for use across the experiments that follow (though it should be noted that not all systems were used in all experiments). The selected systems are further classified based on their sizes, according to the LOC metric, using a classification scheme motivated by the prior work of Zhao and Elbaum (2000), but with changes in its structure in order to meet the growing scale of OSS. Application sizes are therefore categorized into bands based on the number of Kilo LOC (KLOC) in the system:

- **Tiny**: fewer than 1 KLOC
- **Small**: 1 up to 10 KLOC
- **Medium**: 10 up to 100 KLOC
- **Large**: 100 up to 1000 KLOC
- **Extra-large**: comprising more than 1000 KLOC.

[12] http://githut.info

[13] http://stackoverflow.com/research/developer-survey-2015#tech-lang

[14] This does not apply to JDepend, as this particular system was not used in any experiments that involve examining the significance of a correlation between two variables. The system was used only in the experimental work presented in Chapter 5.

Table 3. List of selected OSS

System	Version	URL	Description
JFreeChart	**1.0.17**	http://www.jfree.org/jfreechart	Java chart library that creates a variety of professional quality charts and graphs
FindBugs	**2.0.3**	http://findbugs.sourceforge.net	Static code analyser that analyses Java bytecode to find and detect a wide range of pre-defined bugs and defects. The tool includes more than 200 bug patterns.
JMeter	**2.9**	http://jmeter.apache.org/	An application designed to load test functional behaviour and measure performance of software application
JabRef	**2.9.2**	http://jabref.sourceforge.net	Bibliography tool that provides GUI-based reference management support for *BibTeX* files - the standard *LaTeX* bibliography format.
Apache Commons Lang	**3.3.2**	http://commons.apache.org/proper/commons-lang	Helper utilities for the *java.lang* API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. It contains basic enhancements to *java.util.Date* and a series of utilities dedicated to help with building methods, such as *hashCode*, *toString* and *equals*.
Dependency Finder	**1.2.1-beta4**	http://depfind.sourceforge.net	An analyser that extracts dependencies and dependency graphs of complied Java code and mines some other useful dependency information. The tool also provides basic OO quality metric assessment of source code.
MOEA	**1.17**	http://www.moeaframework.org	A framework that supports development and experimentation of multi-objective evolutionary and optimisation algorithms. The tool is intended to provide fast and reliable implementations of several state-of-the-art multi-objective algorithms.
Barcode4J	**2.1**	http://barcode4j.sourceforge.net	A free and flexible automatic barcodes generator.
JDepend	**2.9**	http://www.clarkware.com/software/JDepend.html	Lightweight analysis tool that evaluates Java packages using several OO quality metrics. The tool provides an automated way to measure the quality of software design.

The aim was to have at least one OSS fit into each of the small, medium and large size categories, as considering systems of different size should enable the applicability of each experimental technique to be assessed at different scales.

It is important to note that, although there have been some efforts to investigate testing in OSS, the actual extent of testing performed and the way developers test their projects in OSS are still unclear. A large scale study conducted on 20,000 OSS projects hosted in GitHub reports that almost 62% of the studied projects have unit tests (Kochhar et al., 2013), although other recent figures from a smaller-scale study (of 460 projects) note that the proportion of *active* tests (i.e., tests that are working and being maintained) is less than 35% (Beller et al., 2015).

Table 4. Characteristics of the selected OSS

System	Total Size (KLOC)	Size	Production Code Size (KLOC)	#Classes (NOC)	Total #of unit tests
JFreeChart	140.6	Large	99.4	669	366
FindBugs	117	Large	114.5	1245	46
JMeter	106.4	Large	90.8	1150	127
JabRef	90.4	Medium	84.7	616	68
Apache Commons Lang	63.6	Medium	23.5	132	142
Dependency Finder	58	Medium	26.7	450	280
MOEA	42	Medium	25.5	407	209
Barcode4J	16.4	Medium	13.2	158	42
JDepend	3.6	Small	2.460	29	18

A short description of each of the nine selected OSS is shown in Table 3. General characteristics of the selected systems are shown in Table 4, where Table 5 provides information about the project's age and the number of developers that contribute to the project's repository. The number of systems used within each experiment is noted in the relevant Chapter (Chapters 5, 7 and 8). Similarly, the numbers of unit tests analysed vary from one experiment to another, depending on the specific objectives of each experiment. Together these systems form around 638 Kilo lines of code (KLOC) and contain in total 4856 production classes and 1298 unit tests.

Table 5. Age and the number of contributors in all selected systems

System	System's age (1st version release date - age)	# Contributors (all releases)
JFreeChart	2000 – 15 Years	8
FindBugs	2007 – 8 Years	11
JMeter	2001 – 14 Years	15
JabRef	2003 – 12 Years	38
Apache Commons Lang	2002 – 13 Years	15
Dependency Finder	2003 – 12 Years	2
MOEA	2011 – 4 Years	1
Barcode4J	2004 – 11 Years	17
JDepend	2003 – 12 Years	2

3.2.2 Statistical Analysis Procedures

A number of statistical analysis procedures are used in this thesis. The selection of procedures again depends on the objectives of the designed experiment, as well as on characteristics of the data being analysed. This section discusses general statistical procedures that have been applied in all chapters.

The Shapiro-Wilk test is used to check whether a data distribution adheres to the characteristics and assumptions of a normal distribution. Shapiro-Wilk is reported to be one of the most powerful normality tests (Razali and Wah, 2011) and it has been recommended for use over other normality tests, such as the Kolmogorov-Smirnov test (Thode, 2002). The Shapiro-Wilk test has been recommended to be used for sample size up to 2000 data points (Royston, 1992, Razali and Wah, 2011). Use of such a test is necessary as selection of statistical analysis tests should be informed by the nature of the distributions being normal or non-normal. Parametric tests assume the data being analysed comes from an underlying distribution, most often the normal distribution, and so the null hypothesis for the Shapiro-Wilk test is that data is normally distributed. Further common statistical analysis procedures and tests are now also explained. Note that for the statistical tests undertaken a threshold of 5% for all obtained significance values (p-values [p]) is employed.

Given the intent to seek evidence of possible relationships among variables, several tests of correlation are conducted across the experiments. Where data distributions are skewed, a rank order test is preferred, so in the experiments reported in this thesis the non-parametric *Spearman's rho (ρ) rank correlation coefficient* test is used. *Spearman's ρ* is a non-parametric procedure to test for statistical association between two independent variables. In some instances the examined correlations are for binary variables (i.e., zero or one, representing absence or presence of a phenomenon). Such data are first classified using the *binary classification* mechanism (through the use of a 2×2 *contingency table*[15]), and then the correlations are examined using the *Phi (ϕ) Correlation Coefficient* test (a statistical measure of association between two binary variables).

Phi correlation is computed from the *contingency table* (shown in Table 6). The *Phi* correlation for variable x and y is calculated using the following formula (1):

$$\varphi = \frac{(a\times d)-(b\times c)}{\sqrt{ac\times bd\times ab\times cd}} \qquad (1)$$

[15] A *contingency table* (or a cross tabulation) is a matrix based table that categorises variables based on their distribution frequency.

Table 6. A 2×2 contingency table

	Y=1	**Y=0**	**Total**
X=1	a	b	ab
X=0	c	d	cd
Total	ac	bd	***n***

Effect size (ez) (for non-parametric data) is measured using the following formula (2) (Fritz et al., 2012, Coolican, 2014):

$$ez = \frac{Z}{\sqrt{N}} \tag{2}$$

where *N* is the number of observations, Z is the standard score (z-value).

Cohen's classification (Cohen, 1988) is used to interpret the degree of association (measured using Spearman's rho, ρ) between variables: there is said to be a *low* association when $0 < \rho < 0.3$, *medium* when $0.3 \leq \rho < 0.5$ and *high* when $\rho \geq 0.5$. This interpretation also applies to negative correlations, but the association is inverse rather than direct (Daniel, 2000). Cohen's classification scheme (Cohen, 1988) is also used to classify effect size values as small ($0 < ez < 0.3$), medium ($0.3 \leq ez < 0.5$) or high ($ez \geq 0.5$).

To compare differences in distribution between two independent groups the non-parametric *Mann–Whitney U* test is used. Where multiple tests are conducted the significance values obtained from the *Mann–Whitney U* test are adjusted/corrected using the Holm-Bonferroni correction procedures (Abdi, 2010) using the following formula (3):

$$p_{\text{H-Bi}} = (C - i + 1) \times p \tag{3}$$

where C is the number of tests conducted; p is the original p-value; i is the rank order of the p-value

A summary of the general statistical analysis procedures used in this thesis is shown in Table 7. The majority of the tests performed in this research are conducted using the *IBM SPSS*[16] (version 22) toolset. Graphs are generated using both *Microsoft Excel*[17] and *R*[18].

Table 7. Summary of the general statistical analysis procedures used in this thesis

Test nature	**Statistical analysis procedure**	**Relevant Chapters**
Measuring the dependence between two variables of non-parametric data	Spearman's rho (ρ) rank Correlation Coefficient test	6 7
Measuring the dependence between two binary variables of non-parametric data	Phi (ϕ) Correlation Coefficient test	7
Compare the significance differences between two independent groups	Mann–Whitney U test	5 7

3.3 Metrics Selection

One of the well-known challenges faced by researchers and practitioners when measuring software products is the choice of appropriate measurements. Metric selection in this research has been determined in a 'goal-oriented' manner using the Goal/Question/Metrics (GQM) framework (Basili and Weiss, 1984) and its extension, the GQM/MEtric DEfinition Approach (GQM/MEDEA) framework (Briand et al., 2002). The GQM concept was first introduced by Victor Basili to encourage selection and use of software metrics in a more systematic manner – as opposed to the prior prevailing convention to measure whatever could be

[16] http://www-01.ibm.com/software/analytics/spss

[17] http://products.office.com/en-us/excel

[18] http://www.r-project.org

measured. The approach was developed originally to support the evaluation of software defects in a set of NASA projects (Basili and Weiss, 1984). GQM is acknowledged as the most widely known and used goal-oriented approach to software measurement (Solingen and Berghout, 1999, Patzke et al., 2012). GQM and other similar goal-oriented frameworks enable researchers and practitioners to transition from what knowledge might be potentially available to that which is actually needed.

The main ***goal*** of this thesis is to better understand what affects software testability, and our ***objective*** is to assess the presence and strength of any relationships between different software characteristics on the one hand and software testability on the other. The specific ***purpose*** is to measure and ultimately predict class testability in OO systems. Our ***viewpoint*** is as software engineers, and more specifically, testers, maintainers and quality engineers. The targeted ***environment*** is Java-based OSS.

The following section presents the general metrics used in this thesis.

3.3.1 Source Code Metrics

Program size is measured using the two well-known static metrics: LOC[19] and Number of Classes (NOC). Class size is measured in terms of the Number of Methods (NOM) within a class and the number of LOC within that class. We also measure the complexity of a class using the class's Cyclomatic Complexity (CC). Dynamic Coupling is measured using the Dynamic Coupling Between Objects (DCBO), Import Coupling (IC) and Export Coupling (EC) metrics (dynamic coupling metrics are explained in more details in Sections 5.5 and 6.2.1, respectively). Details of the selection of these dynamic metrics and their specialised data collection procedures are provided in Chapters 5 and 6.

[19] LOC counts the number of all lines other than blank and comments-only lines. This also applies to the TLOC metric.

3.3.2 Class Testability Metrics

Testability in this study is considered at class-level. As explained previously, class testability refers to the one-to-one relationship between each production class and the corresponding unit test that is designed to test the production class. Following the ISO definition of testability, we *define* class testability in terms of the attributes of a class that bear on the *effort needed* to validate the class. Class testability is then *measured* in terms of the size of the unit test. Two static metrics are used for this purpose: the Test Lines of Code (TLOC) and Number of Test Cases (NTC). These metrics are motivated by the test suite metrics suggested by Bruntink and van Deursen (2006). TLOC, derived from the classic LOC metric, is a *size* measure that counts the total number of physical lines of code within a test class. NTC is a *design* metric that counts the total number of test cases in a unit test. These same testability metrics have been widely used in several previous studies (e.g., (Bruntink and van Deursen, 2006, Badri et al., 2010, Zhou et al., 2012)). Note that the two class testability measures are themselves known to be correlated i.e., TLOC increases with NTC.

Because the two metrics are collected at post-production phase, they may in fact represent the effort expended to test a class. The assumption here is that the effort expended is indicative of the effort needed. This has a reflection on the test suites of the OSS we used in the thesis. We conduct some of the analysis of this thesis (especially in Chapters 6 and 7) under the assumption that the two class testability metrics are representative of the effort needed to test a class, leading us to make some conclusions about the relationship between several class artefacts and class testability.

We examined the suitability of the test suites to ensure that the set of unit tests we are using here are representative of the effort needed to test production classes. We first check the relationship between unit tests size and production class size in all examined systems. Size has been widely used as an indicator of many aspects of software development processes. For instance, well-established effort estimation models such as the COnstructive COst MOdel (COCOMO) (Boehm, 1981) and the Software LIfecycle Management (SLIM) model (Putnam, 1978) use size (and specifically LOC) as the main input to their models. In keeping with such models, in this study there is an expected relationship between the effort required to develop a class and the effort required to test the class. In other words, there is an expected strong relationship between the size of a class and the size of the unit test that is designed to test the same class. (Such a relationship has been shown in previous research (Bruntink and van Deursen, 2006).) We checked if such a relationship existed in the systems we examined in this thesis, to get an indication whether the effort expended to develop a production class is related to the effort expended to test the class. We therefore statistically examine the correlation between LOC and TLOC in all eight systems using the *Spearman's rho* test (see Section 3.2.2). Detailed results of the correlations are shown in Table 8.

The results of this test showed that both metrics are significantly correlated in all but one of the examined systems (with FindBugs being the exception (showing no significant correlation)). There is a strong positive correlation between the size of a class and the size of its associated unit test in six systems (JFreeChart, JabRef, Commons Lang, Dependency Finder, MOEA and Barcode4j); and one system showed a medium, significant correlation (JMeter). In considering size as a proxy for effort, these relationships suggest that the more effort that is needed to write a class, the more effort will be expended to test that class.

Table 8. Spearman's ρ correlations between LOC and TLOC

		LOC							
		JFreeChart	**FindBugs**	**JMeter**	**JabRef**	**Commons Lang**	**Dependency Finder**	**MOEA**	**Barco-de4J**
TLOC	ρ	**.50**	-.11	.30	.68	.71	.56	.52	.52
	p	**.00**	.26	.01	.00	.00	.00	.00	.00

A second task was carried out to check the extent of the test suites' coverage – high coverage provides some assurance of the completeness of the testing (though it is acknowledged that *high* coverage does not necessarily indicate 'high quality' tests[20]). We measure four different types of coverage: *statement coverage* (number of executed (tested) statements), *branch coverage* (number of executed control structures (such as *if-else* and *case* statements), *line coverage* (number of executed lines of code) and *method coverage* (number of executed methods). Descriptive statistics of the retrieved coverage information are provided in Table 9.

Table 9. Coverage infromation from a selected set of classes

System		Statement Coverage	Branches Coverage	Line Coverage	Method Coverage
JFreeChart	**Mean**	58%	89%	60%	65%
	Median	60%	51%	65%	63%
	Std dev.	0.275	3.189	0.260	0.228
FindBugs	**Mean**	51%	37%	52%	57%
	Median	46%	29%	43%	49%
	Std dev.	0.285	0.348	0.276	0.270
JabRef	**Mean**	77%	70%	77%	82%
	Median	98%	84%	96%	100%
	Std dev.	0.323	0.342	0.316	0.280
Commons Lang	**Mean**	87%	79%	88%	89%
	Median	96%	82%	97%	100%
	Std dev.	0.216	0.232	0.209	0.216

We collected test coverage information for a sample of classes over a number of the systems investigated in detail in Chapters 6 and 7. Specifically, we examined a selection of classes from two of the large-size systems (JMeter and JFreeChart) and two of the medium-size systems (JabRef and Apache Commons Lang). In total we examined 20% of the unit tests (and their associated production classes) from each

[20] http://martinfowler.com/bliki/TestCoverage.html

of the abovementioned systems. The 20% of classes were selected based on the size of the unit tests – half of these (10%) are selected from the 'smallest' unit tests in each system, while the other half (10%) are selected from the set of 'largest' unit tests. In total, we examined the test coverage of 72 unit tests from JFreeChart, 8 from FindBugs, 12 from JabRef and 24 from Commons Lang.

The results (Table 9) show that all of the investigated classes exhibited fair to high levels in regard to *method coverage* (between 57% and 89%) and *branch coverage* (between 37% and 89%). Branch and method coverage are relatively high in JFreeChart, JabRef and Commons Lang. Taken together, the results of the correlation analysis between class size and unit test size and the analysis of unit test coverage provide reasonable evidence that an appropriate level of effort had been expended when constructing the test suites. We therefore believe that the unit tests used in the experiments that follow are indeed generally representative of the effort needed to test the targeted production classes.

3.3.3 Code and Test Smells

All code and test smells are detected using automated tools, and some of the smells are identified using a metrics-based approach. Details of the test and code smells considered in this thesis are provided in Chapter 7.

3.4 Data Collection Methods and Procedures

To identify unit tests and associate them with their corresponding production classes in the selected OSS, two established test-to-code traceability techniques are used (Van Rompaey and Demeyer, 2009). First, we used the *Naming Convention* technique, which reflects the widely suggested practice (for instance, in the JUnit documentation) that a unit test should be named after the corresponding class that it tests, by adding "Test" to the original class name. For example, the unit test for class 'Domain' should be: 'DomainTest'. Second, we used a *Static Call Graph* technique, which inspects method invocations in the test case. Both processes were carried out manually. The effectiveness of the *Naming Convention* technique is reliant on developers' efforts in conforming to the recommended coding standard, whereas the *Static Call Graph* approach reveals direct references to production classes in the unit tests. Van Rompaey and Demeyer (2009) explained that traceability strategies such as *Naming Convention* results in high precision and recall but also they depend on the testing strategy and guidelines followed. Other strategies such as *Static Call Graph* and *Last Call before Assertion* have high applicability but score low in accuracy. Therefore, the authors recommended using a combination of strategies, which we did here.

It is important to note here that only core system code is considered: that is, only production classes that are developed as a part of the system are assessed. Additional classes (including those in jar files and external libraries) are excluded from the measurement process. These files are generally not part of the core system under development and any dependencies could negatively influence the results of the measurement process.

A set of tools and procedures has been used to collect and analyse the data. A summary of the tools used in this thesis, and in which specific chapters their use is reported, are noted in Table 10. In terms of the metric collection tools employed, we selected tools that follow the same metric definitions that we identified for all static metrics. For example, we defined LOC as the number of all lines other than blank and comment-only lines in class. Therefore, we sought automated tools that measured LOC following the same definition. The same applies to all other static metrics of interest, such as NOM, CC and TLOC.

All data (i.e., metrics) collection and access to the OSS were performed in *Eclipse*. All static metrics, including LOC, NOC, NOM, CC and TLOC, are collected using the *CodePro Analytix* tool. The values of these metrics were later checked and verified using the *Eclipse Metrics Plugin*. Values for the NTC metric are collected directly from the *JUnit* framework and these values were verified manually.

Table 10. List of Tools Used in the Thesis

Tool	Description	Usage	Chapter
Eclipse IDE[21]	A multi-language integrated development environment	Development and access to OSS. Most of the tools listed below have *plugins* that integrate well with Eclipse	5,6,7
AspectJ[22]	An AOP implementation for Java language	Collection of the dynamic metrics data	5,6,7
JUnit[23]	Unit testing framework for Java	Analysis of unit test suites	5,6,7
CodePro *Analytix*[24]	Java source code analytical tool	Collection of a set of static metrics	5,6,7
InCode[25]	An industrial code smell and design flaw detection tool	Collection of a set of code smells	7
JDeodorant[26]	Code smell detection and refactoring tool	Collection of a set of code smells	7
PMD[27]	Static Java source code analyser	Collection of a set of code and test smells	7
Eclipse Metrics Plugin[28]	Static metrics tool for Java code	Collection of a set of static metrics	5,6,7
NodeXL[29]	Template for Microsoft Excel that creates network graphs	Creation of dependency graphs and extraction of graph metrics	5
Emma[30]	Code coverage tool	Measurement of several test coverage levels	5,6,7
CodeCover[31]	Code coverage tool	Measurement of several test coverage levels	5,6,7

[21] https://eclipse.org

[22] https://eclipse.org/aspectj

[23] http://junit.org

[24] https://developers.google.com/java-dev-tools/codepro/doc

[25] https://www.intooitus.com/products/incode

[26] http://www.jdeodorant.com

[27] http://pmd.sourceforge.net

[28] http://metrics2.sourceforge.net

[29] http://nodexl.codeplex.com

[30] http://emma.sourceforge.net

[31] http://codecover.org

3.5 Threats to Experimental Validity

There are a number of threats that can affect the results presented in this thesis. Some of these validity threats are common threats that could impact the results of two or more experiments – these are therefore considered here. Details of the specific validity threats that could affect each individual experiment are presented in the relevant Chapters. Note that the validity threats to the mapping study (Chapter 5) are different in nature than those relevant to the experiments presented in Chapters 6, 7 and 8.

3.5.1 Internal Validity

Ambiguity about Direction of Causal Influence

This refers to the question of which variable causes or influences the other (e.g., A causes B, B causes A, or even X causes A and B). In some of the analyses that follow we build in assumptions of directionality (impact or cause-effect) in the correlations between different variables based on the theory and findings identified in previous research. However, we do not investigate the directed impacts between individual variables *per se*. Further specific details of this internal validity threat are presented in Chapters 6 and 7.

Confounding Variables

Some of the correlations identified between variables might be influenced by confounding factors (e.g., A causes X which causes B). A confounding factor is an unmeasured mediating factor, whose presence means it is difficult to distinguish the effects of factors on each other (Wohlin et al., 2012). These confounding factors might be the reason why some of the discovered correlations appear. Thus, although we study the strength of correlation between different variables, in the experiments reported here we ignore the possibility that there are other factors that could cause or influence the discovered correlations. The influence of confounding variables is discussed further in Chapter 7.

Selection of Executions Scenarios

This threat applies specifically to the experiments in Chapters 5 and 6. Execution scenarios are designed to mimic as closely as possible 'actual' system behaviour, based on the available system documentation and, in particular, indications of each system's key features. However, it is important to acknowledge that the selected scenarios might not be fully representative of the typical uses of the systems. Analysing data that is collected based on different scenarios might give different results. This particular issue is very common in most research that employs dynamic analysis techniques. However, this threat is mitigated by carefully checking user manuals and other documentation of each of the examined systems and deriving the chosen scenarios from these original sources. Most listed features were used (at least once) during the execution.

Class Testability Metrics

Class testability is measured in this thesis using two unit tests metrics i.e., TLOC and NTC. As explained in Section 3.3.2, these metrics have been used in several previous works to measure class testability. However, if best practice is not followed then the two metrics may not necessarily correspond to class testability. For example, these metrics do not take into account the number of test cases that are actually required to effectively test a production class. Rather, these metrics reflect the true development practice in the examined OSS. To reduce the impact of this threat we decided to use OSS that have been in use (and under continuous improvement) for several years (see information about systems' ages in Table 5). We chose the latest available version of each system at the time when they were selected. Furthermore, we also included well-known systems (such as JFreeChart, FindBugs and JabRef) and systems from well-established open source groups/foundations such as the Apache Software Foundation (i.e., JMeter and Commons Lang). Therefore, given the maturity and long-term use of the selected systems, we would expect that they would have adequate test suites, following best practice. This threat applies to the experiments in Chapters 6 and 7.

Selection of Classes and Unit Tests

Production classes and unit test selections can be another validity threat. Only production classes that have corresponding unit tests are included, which may lead to a selection bias. Since none of the examined systems have available unit test for all production classes (i.e., provide a complete class coverage), classes that are extremely difficult to test, or are considered too simple, might have no associated unit test and, hence, would not be considered in our analyses.

3.5.2 External Validity

Generalisation of Findings

A total of nine different OSS are examined in the course of the research conducted for this thesis. The number of examined systems varies from one experiment to another, however. While in each experiment a minimum of four systems is used, and those systems have been selected with deliberate criteria in mind and to cover a range of system sizes (see Section 3.2.1), there is still a possibility that consideration of other systems might affect the results obtained, and that more systems are needed to validate the results for the purpose of wider generalisation. This threat applies to all three experiments (Chapters 5, 6 and 7).

Limited Scope of System Coverage

This thesis presents analysis that has been conducted on OSS applications due to the availability of their source code (both production code and unit tests). Therefore, the results of this study are limited to OSS and cannot be directly generalised to include closed-source, industrial applications. Although we have used a variety of systems in this study (i.e., applications of different sizes and from different domains), we acknowledge that experiments across more applications will build greater assurance regarding our current conclusions. This threat applies to all three experiments (Chapters 5, 6 and 7).

Efficacy of Tools Used

All metrics used in this research are collected using automated tools (see Section 3.4). The accuracy of these tools – or lack thereof – could be a threat to the findings. To address this threat we selected tools that have been used in previous research in empirical software engineering. Where possible, data collected by one tool was cross-validated by those obtained from a similar tool[32]. (For example, static metrics data were collected using CodePro Analytix (see Table 4) and were validated using the Eclipse Metrics plugin.) Some of the tools are open source, free-access tools (such as *CodePro Analytix*), while others are closed-source, industrial-based tools (such as *InCode*).

3.5.3 Conclusion Validity

Selection of Statistics Methods

The use of statistical analysis procedures in this work is central to its outcomes. Therefore, the decisions made regarding the selection and application of all statistical methods used in this research must be robust and appropriate. In all cases the analyses are driven by clearly defined research questions and hypotheses. The specific statistical analysis methods used are selected based on well-known reference works in statistics and on the conduct of previous similar research in empirical software engineering. Moreover, the selected tests (especially for the experiment used in Chapter 7) were further validated through discussions with two academic statisticians from the Department of Mathematics and Statistics, University of Otago.

[32] Note that using multiple tools was not possible in many cases due to the availability of the tools.

3.6 Summary

This Chapter has described the research methodology that applies across the work reported in the thesis. It explains the general experimental design, the systems used in the experiments, the data collection methods and the statistical procedures used to analyse the data, as well as possible general validity threats.

The following Chapter presents the results of a systematic mapping study on dynamic metrics and their use in measuring software quality.

Chapter 4 A Systematic Mapping Study on Dynamic Metrics

4.1 Introduction

After early advances in the adoption of systematic measurement programmes in the 1980s software measurement went out of favour in industry. Organisations saw measurement as a costly overhead that was delivering limited, if any, benefit. This was due to over-zealous promotion of the use of software metrics as a panacea to the major challenges of software project management, poor alignment of measurement programmes with organisational goals and priorities, and high compliance costs. The relatively recent emergence of analytics as a real-time component of organisational intelligence and improvement, however, has seen a resurgence in interest in software measurement – 'rebranded' as software analytics. Managers are once again seeking efficient ways to continually evaluate and improve their software processes and products. A renewed research focus has therefore been directed to the use of software metrics in helping software engineers to measure and assess various characteristics of the software components they produce. As introduced in Chapter 2 (Section 2.3), there are two different sets of software metrics that might be useful: static metrics and dynamic metrics. The latter is the subject of the systematic mapping study presented in this chapter.

A systematic mapping study is intended to deliver a comprehensive summary of the body of research related to a specific topic, based on primary studies identified through a robust search strategy. As such, mapping studies tend to cover a wider spectrum of work in comparison to systematic literature reviews. Once found, the body of prior work related to a topic area is then categorised and quantitatively characterised based on a variety of dimensions (Kitchenham et al., 2011). Systematic mapping studies are mainly concerned with structuring a research area under investigation, whereas systematic literature reviews aim at synthesising evidence (Petersen et al., 2015). Results from systematic mapping studies can help to highlight issues that could benefit from further investigation via new primary studies, or new research perspectives that could be brought to bear on the topic. (Differences between systematic mapping and review studies are covered in Kitchenham and Charters (2007), Petersen et al. (2008) and Kitchenham et al. (2011)).

This chapter presents the results of one such mapping that sets out to identify and classify relevant research on the topic of dynamic software metrics. The objectives of this review are to obtain a general overview of the prior research conducted on this topic and to inform researchers and other readers of potential research gaps that could be studied further. In terms of its role in this thesis, the particular motivation behind this review is to systematically investigate which dynamic metrics have been used to measure aspects of software quality. Dynamic software metrics, their key advantages and the techniques used to the collect them have been explained in detail in Chapter 2 (Section 2.3.2). As also noted, it is a goal of this research to use dynamic metrics to measure software testability.

As far as can be ascertained there has been just one prior review on the topic of dynamic metrics (Chhabra and Gupta, 2010), and this was not a systematic review. In that work Chhabra and Gupta (2010) summarised the research problems, challenges and opportunities relevant at that time in the dynamic software metrics domain. The paper discussed some of the most notable works in the field of Object Oriented (OO) metrics, such as the 'CK' metric suite (Chidamber and Kemerer, 1994) and the MOOD metrics suite (Harrison et al., 1998). While undoubtedly informative, the review was informal; i.e., there were no defined research questions, search process or data extraction process. Perhaps as a result, the review overlooked some important works in the area of dynamic metrics, including those of Dufour et al. (2003a), (2003b) and (2004) that addressed dynamic metrics for Java programs and compiler developers, and the *AspectJ* tool, and the early work of Voas (1992) on software reliability.

Initial observations of the relevant body of literature indicates that the first paper to study dynamic metrics is indeed that of Voas (1992) just noted. (Note that this study does not consider other forms of run-time measurements such as those concerned with system performance and other time-dependent measurements. These reflect aspects of systems not relevant to maintenance or re-engineering effort, and as such, we consider such work to be outside the scope of this thesis.) This work proposed the *Revealing Ability* metric, a dynamic metric that predicts a program's ability to allow faults to be undetected during dynamic testing. Another work published in the same year, by Munson and Khoshgoftaar (1992), used a unique run-time metric called *Functional Complexity* to measure the dynamic complexity of a software system. The publication of these papers signalled the beginning of the research effort on the topic of dynamic software metrics.

The review methodology used in this systematic mapping study is detailed in the following section.

4.2 Review Methodology

This review targets the topic area of dynamic software metrics. It is intended to characterise and evaluate the use and utility of dynamic metrics, identifying the benefits and drawbacks of this group of software metrics as described in the body of literature. This work was conducted according to the guidelines suggested by Kitchenham and Charters (2007) and informed by the mapping study-specific guidelines of Petersen et al. (2008).

The remainder of this Section discusses the research questions, search strategy and search process. The review protocol, which provides a high-level overview of the steps in the review study, is shown in Figure 4

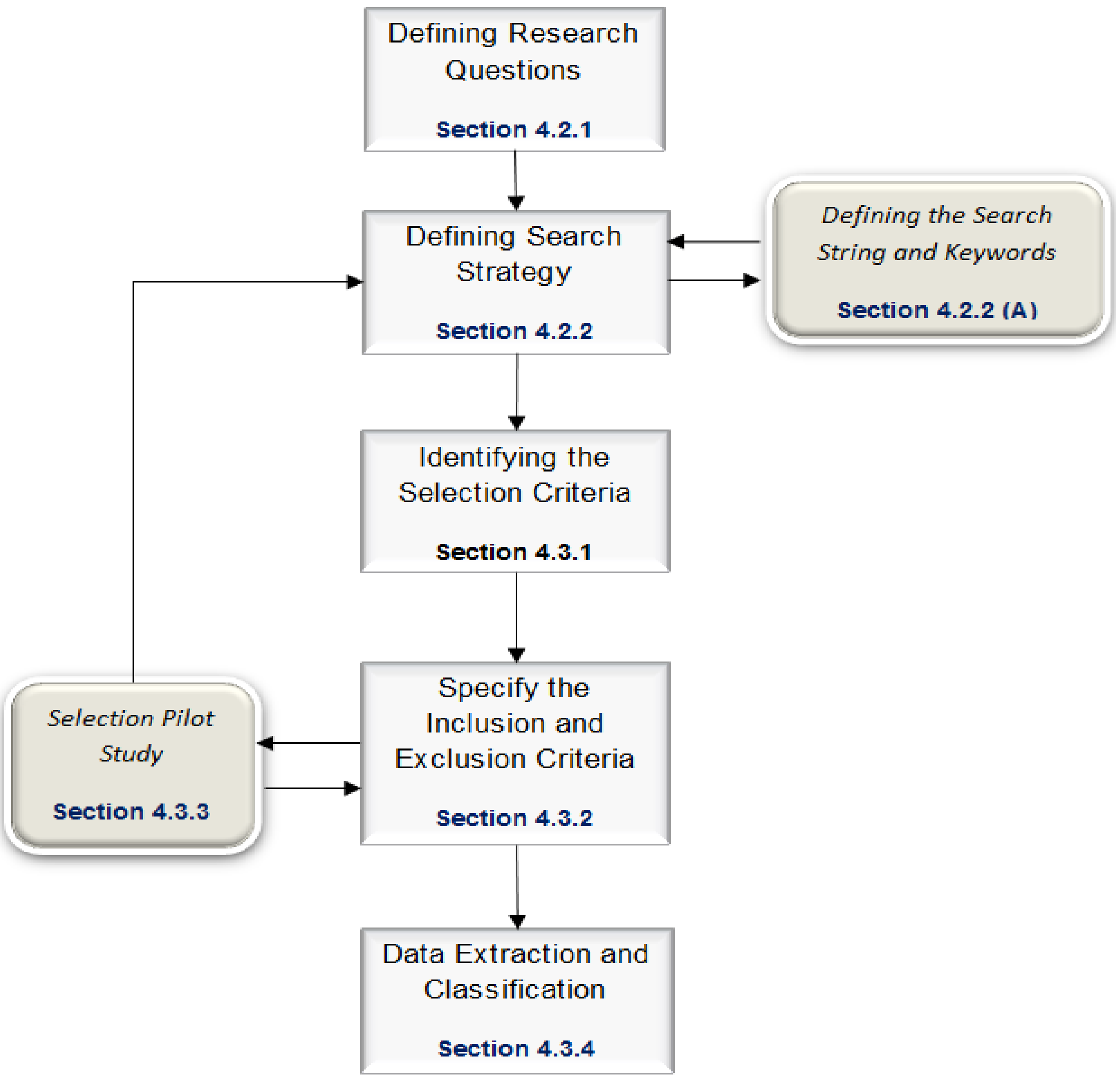

Defining Research Questions
Section 4.2.1
Defining Search Strategy
Section 4.2.2
Defining the Search String and Keywords
Section 4.2.2 (A)
Identifying the Selection Criteria
Section 4.3.1
Specify the Inclusion and Exclusion Criteria
Section 4.3.2
Selection Pilot Study
Section 4.3.3
Data Extraction and Classification
Section 4.3.4

Figure 4. Review protocol

4.2.1. Research Questions

For any review work, defining the set of research questions is a critical first step, as the research questions directly inform the search and data extraction strategies. The two research questions for this systematic mapping study are:

RQ$_{4.1}$: Which aspects of dynamic metrics have been most frequently subjected to study?

RQ$_{4.2}$: Which aspects of dynamic metrics could be recommended as topics for future research?

RQ$_{4.1}$ addresses recent and current research addressing the use of dynamic software metrics, categorizing all research activities in the field within a defined time window. In terms of advancing the field it is essential to have a baseline as to what metrics have already been developed and an understanding of the characteristics that these metrics actually measure. Answering this question should also contribute to an understanding of the usefulness and the drawbacks of this group of metrics. By fully understanding the metrics, their coverage and their mechanisms of action, we should then be in a position to identify any current difficulties or limitations associated with their use, as a precursor to suggesting solutions or possible avenues of further primary investigation – thus informing RQ$_{4.2}$. RQ$_{4.2}$ is expected to be of help in directing future research in the field, based on stated, and implied, research limitations, open problems and newly identified gaps.

4.2.2. Search Strategy

The search process is divided into two main phases: *Automatic* and *Manual*. The Automatic search is used to search for materials via electronic search engines using a defined (and pre-tested) search string. The Manual search, on the other hand, is performed by researchers scanning and reading through selected journals and conference proceedings manually. This step can help to assure coverage of a wider range of materials, enabling verification of the efficacy of the automatic search and helping to ensure that the review does not miss relevant primary studies in the literature. Our search was conducted for the period between January 1992 and December 2014. As stated in Section 4.1, initial evidence suggests that the first two papers to study dynamic metrics were both published in 1992. In order to confirm the appropriateness of the selected search period, a brief search was conducted in IEEEXplore database using the term *"dynamic metrics"* seeking articles published in the two years before the suggested starting date of our search period (i.e., between 1990 and 1992). No related articles were found. Thus, it was confirmed that the search period should begin from January 1992.

A) Automatic Search

The automatic search was conducted using two different electronic sources, namely: *SCOPUS* and *Google Scholar*. *SCOPUS* has a user friendly search engine that provides efficient and complete web access to over 5,000 international publishers as well as hundreds of open access journals. *SCOPUS* indexes well-known publishers that publish papers in computer science and information technology, including: *IEEE*, *ACM*, *Elsevier*, *Springer* and *Wiley-Blackwell* publishers. In addition, *Google Scholar* is used to reveal technical reports and articles that could not be found by *SCOPUS* (e.g., have not been published by the abovementioned publishers). *Google Scholar* is a powerful search engine that provides very wide coverage of articles and materials on the web.

The search string that is used in the automatic searching process is shown in Figure 5. It has been noted previously that using a specific and verified search string (see Section 4.3.3) may improve the search process by increasing the likelihood of finding relevant studies while reducing search workload (MacDonell et al., 2010).

```
((software OR program) AND ("dynamic metrics" OR "dynamic
metric" OR "dynamic measurement" OR "runtime metrics" OR
"dynamic measure")) OR ("dynamic analysis" AND ("program
comprehension" OR "program understanding") AND metrics)
```

Figure 5. Search string

Table 11. Manual search results

Type	Names	Acronym	No of Retrieved Articles (filtered by title)	No of Selected Papers (filtered by abstracts)	No. after Removing Duplications
Journals	IEEE Transactions on Software Engineering	TSE	15	6	3
	ACM Transactions on Software Engineering and Methodology	TOSEM	0	0	0
	Journal on Systems and Software	JSS	14	6	6
	IEEE Software	IEEE Softw.	2	2	2
	Information and Software Technology	IST	6	3	3
	Journal on Software Maintenance and Evolution	JSME	7	2	1
	Empirical Software Engineering Journal	ESEJ	10	4	2
	Software Quality Journal	SQJ	4	2	1
			58	**28**	***18***
Conferences	International Conference on Software Engineering	ICSE	9	7	7
	International Conference (Workshop) on Program Comprehension	ICPC (IWCP)	13	13	13
	IEEE International Software Metrics Symposium	METRICS	4	4	2
	International Conference on Software Maintenance	ICSM	12	8	5
	Workshop on Program Comprehension through Dynamic Analysis	PCODA	6	5	5
	International Symposium on Empirical Software Engineering	ISESE	0	0	0
	International Symposium on Empirical Software Engineering and Measurement	ESEM	4	3	2
	Workshop on Dynamic Analysis	WODA	2	0	0
	Workshop on Emerging Trends in Software Metrics	WETSoM	2	2	1
			52	**42**	***35***

This search string was used only with the *SCOPUS* database portal. Unfortunately, the nature of Google Scholar's search structure did not support us effectively using our search string as defined. When it was attempted to do so it returned a huge, unworkable number of papers and materials, many of which were not even related to the field. Therefore, it was decided to use a much simpler search string term, "*Dynamic Metrics*", to search for papers via Google Scholar. It is important to highlight that Google Scholar was used here mainly as a secondary source to improve the level of assurance regarding coverage of the relevant literature.

B) Manual Search

Unless an automatic search string is extremely obscure it is basically a given that such a search will find more results in comparison to a necessarily labour-intensive manual search. However, if they are not conducted with care, automatic searches can be of poor quality (Kitchenham et al., 2010). The value of a manual search is in increasing the reliability of the search process, through increased assurance that important literature in the field that cannot be found using the search string is not missed in the review. The overlooking of studies in automatic searchers occurs mainly due to restriction criteria on the scope of automatic searches. Combining the two techniques, automatic and manual, can thus solve problems that might arise when using either the manual or automatic search only.

During this phase we searched manually for relevant articles in a list of eight journals and nine conference and workshop proceedings relevant to the defined research topic. Based on our prior knowledge of the research domain, these journals, conferences and workshops were known to be closely related to software metrics, program analysis and comprehension, and the software maintenance and reengineering fields. The retrieved articles were filtered based on titles and then abstracts. The full list of the selected journals and proceedings traversed in the manual search is shown in Table 11.

C) Reference Checking

In addition, the lists of references of all the selected articles relevant to our study were examined. This additional process can add to the coverage of the literature and further reduce the chances of omitting any significant work in the field (Cornelissen et al., 2009). Potentially relevant studies identified through this process are added to the selection loop so that they too are examined for relevance in terms of the review and their adherence to the defined selection criteria.

4.3 Search Process

The review process was composed of six main stages. Figure 6 highlights the review process and the numbers of publications identified by the end of each stage.

As shown in Figure 6, the automatic and manual searches are conducted in stage one. The process began with the automatic search of the *SCOPUS* database (which found 487 papers) and was followed by the manual search of the list of journals and proceedings noted in Table 11. In stage 2, the first filtration was performed by discarding papers with irrelevant titles that had been returned by the automatic search, leaving 97 papers. In stage 3, articles were filtered based on their abstracts leaving 74 papers, although when the abstract was ambiguous or unclear, the introduction was also checked. The latter two stages (stages 2 and 3) were not needed for the manual search, as selection in the first place was based on the title and abstract. This was followed by the second automatic search, this time on *Google Scholar*, which resulted in the identification of 20 potentially relevant articles (filtered by titles and abstracts). In stage 4 of the search process, the results of both the automatic and manual searches are then combined, giving 95 studies. A full text review of those studies was then performed in stage 5, leaving 64 papers. Finally, a reference check was then conducted on all selected papers in the final list. During this step, 20 additional papers were retrieved, and 8 of these were added to the final list of selected articles and another round of reference checking was conducted on the newly added papers. Inclusion and exclusion criteria (Section 4.3.2) were applied through all stages. The final list included 69 studies.

4.3.1 Study Selection Criteria

Based on the goals of the review and the identified research questions, a set of selection criteria is defined. The selection criteria of the identified studies are as follows:

- All works must be relevant to dynamic software metrics. The main goal is to select works related to dynamic software metrics and their applications. This may exclude related but separate topics as tracing, debugging, and program slicing.
- The review strongly focuses on dynamic metric topics (processes, techniques, methods and tools); works that only address static metrics will not be considered.
- This work studies dynamic metrics and their collection techniques. Studies such as those focuses on the use of dynamic analysis in trace visualisation are not considered.

4.3.2 Inclusion and Exclusion Criteria

Inclusion and exclusion criteria are used to filter and rule out studies that are not relevant to our defined research questions. The review included papers published between January 1992 and December 2014. Only primary studies were included; secondary studies (such as review studies) are excluded.

We also exclude the following:

· Papers not written in English.

· Editorials, prefaces, covers, article summaries, books, interviews, news, correspondence, discussions, comments, tutorials, readers’ letters and summaries of workshops and symposia.

- Duplicated studies (e.g. several reports of the same study published in different places or on different dates).
- Studies that did not specify which particular dynamic metrics were used.

4.3.3 Selection Pilot Study

It has been suggested that those undertaking a review of this nature should conduct a pilot study to validate the selection approach and verify the effectiveness of the search string before conducting the actual review (Kitchenham and Charters, 2007). Thus a pilot study was conducted using a short search string (*software* AND *dynamic* AND *metrics*) to search for materials in the IEEEXplore database. We searched for articles between January 2001 and January 2011. Based on the pilot automatic search, the total number of articles retrieved was 298. After applying the defined inclusion and exclusion criteria (Section 4.3.2*)*, 22 relevant articles were selected. The results of this pilot study were validated by checking whether the pilot search returned articles that are known in the field. Fifteen of the 22 articles were identified as being familiar, and the other seven were deemed to be relevant based on a review of their content. Based on these results, we concluded that these terms should be included in the search string, as they returned relevant articles. The list of the selected papers from the pilot study is found in Appendix A.

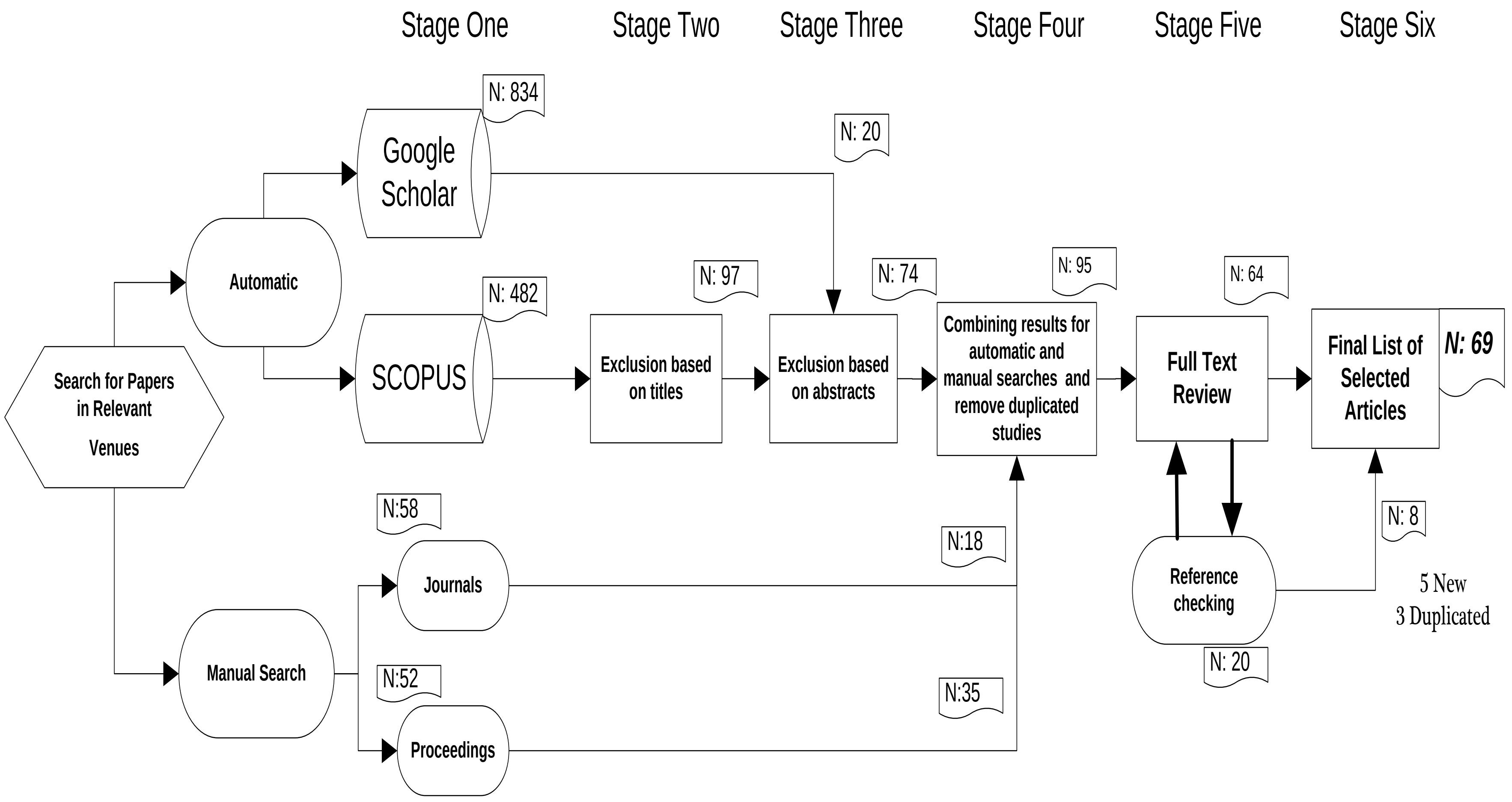


Figure 6. Overview of the mapping review process

4.3.4 Paper Classification Schemes

The general classification scheme proposed by Petersen et al. (2008), in their foundation work on systematic mapping studies in software engineering, is used to classify the retrieved studies. Publications were categorized in three distinct classification schemes: research type, research focus and contribution type. For the research type, the categorization scheme proposed by Wieringa et al. (2006) is used. This scheme was recommended by Petersen et al. (2008) and has been used in several recent systematic review and mapping studies, such as that of Kitchenham et al. (2011). Research type is categorized into the following categories, as suggested by Wieringa et al. (2006) and summarised by Petersen et al. (2008):

1) **Evaluation papers**: evaluate the use and implementation of different problems, techniques or solutions. It shows how the technique or method was implemented and what the consequences of implementation are.
2) **Proposal papers**: solutions are proposed. It also argues for the relevance of the proposed solution without any providing any in-depth analysis. The proposed solution must be novel, or a significant improvement of an existing one.
3) **Validation papers**: investigate novel techniques or methods that have not been implemented previously.
4) **Philosophical papers**: propose a new way of looking at existing things.
5) **Opinion papers**: express the personal opinion on certain techniques or methods. Opinion papers do not usually rely on related literature or specific research methodology.
6) **Experience papers**: focus on the personal experience of the author on specific matter (e.g. project). Experience papers must contain some of the lessons learned in practice by the author(s).

Contribution type is classified into five main categories as follows:

1) **Method**: description of how to measure specific software aspects.
2) **Process**: research that deals with the measurement process itself.

3) **Tool**: any automated tool that is designed to support the measurement process (in the form of a prototype).
4) **Metrics**: any metrics designed to measure aspects of software programs (both new metrics and claimed improvements on existing metrics).

Finally, research focus is classified into four main categories:

1) **Estimation:** metrics that are used for the purpose of estimation (e.g. size or complexity estimation metrics).
2) **Design level**: metrics that can be collected at the design level or early in the development process (e.g. metrics that may be collected from UML diagrams).
3) **Code level:** metrics that are related to the source code level (e.g. code complexity and size metrics).
4) **Reengineering/comprehension:** metrics that are used for the purpose of reengineering, comprehension, understanding or maintenance. Some of these metrics can also be related to the design or code level, but here we consider them separately from the other groups.

4.4 Results

After determining all of the relevant articles the total number of selected primary studies was 69. (The numbers of articles found using the manual search were shown in Table 11.) The distribution of the selected studies (Table 12) shows that the majority were published in conference proceedings. Table 13 shows the distribution of articles per publication venue.

Table 12. Distribution of articles per source type

Publication Type	No. of Studies	Percentage
Journals	19	27%
Conferences	38	54%
Workshops	9	13%
Technical Reports/ Newsletters	4	6%

As shown in Figure 7, it is evident that the number of publications addressing dynamic metrics has not been particularly steady since 2002, and that 41 of the 69 articles (59%) were published between 2007 and 2014.

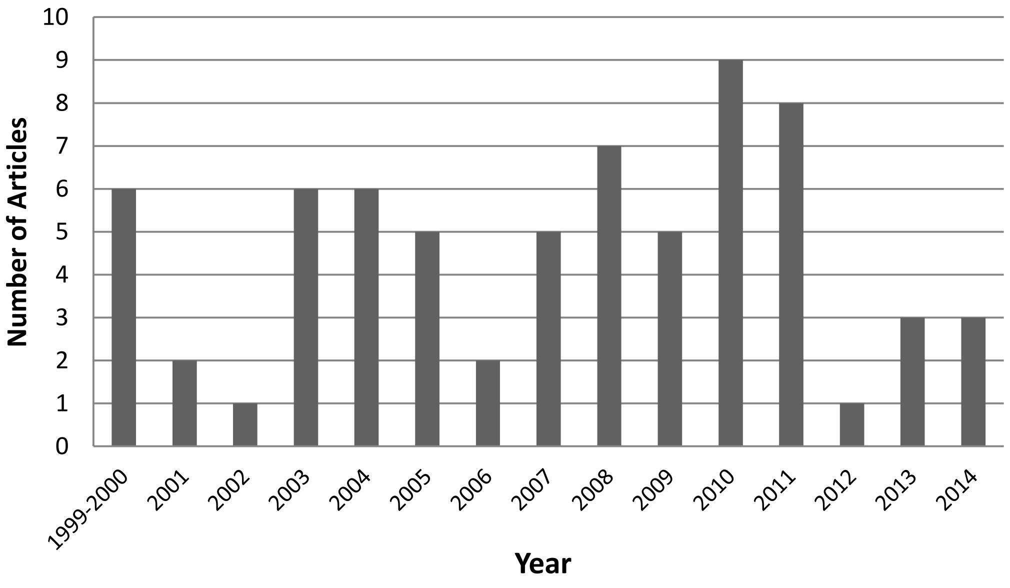


Figure 7. Articles distribution per year

In order to provide a more accessible representation of the extracted results we chose to summarise the data using tables and visual representations. Figure 8 depicts a map of publications over the defined classification criteria. Research focus is shown on the *Y* axis, Contribution type is shown on the right *X* axis, and Research type is shown on the left *X* axis. Each bubble's size represents the number of publications in the corresponding category pair. As is evident, the proposal and evaluation of code metrics and methods currently dominate the body of literature on this topic. Breaking Figure 8 down, Figures 9, 10 and 11 show the distributions of articles per research type, contribution type and research focus respectively.

Table 13. Top publication venues

Conference Name	Abbreviation	Type	No. of articles
ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications	OOPLSA	Conference	3
IEEE Transactions on Software Engineering	TSE	Journal	2
International Workshop/Conference on Program Comprehension	IWPC/ICPC	Workshop/ Conference	2
ACM Software Engineering Notes	ACM SIGSOFT	Newsletter	2
International Conference on the Principles and Practice of Programming in Java	PPPJ	Conference	2
IEEE International Working Conference on Source Code Analysis and Manipulation	SCAM	Conference	2
Journal of Systems and Software	JSS	Journal	2
Workshop on Program Comprehension through Dynamic Analysis	PCODA	Workshop	2

We also considered classifying metrics in terms of the software engineering/programming paradigm that they belong to (Figure 12). Based on the retrieved data, it was found that a high proportion of the papers dealt with OO metrics: over 78% of the studies (Figure 12). The remainder addressed a mix of procedural, aspect-oriented and service-oriented approaches. Studies of specific named metrics were categorized based on their use. It is important to note that many of the metrics and factors are interrelated. For example, coupling metrics are related to complexity. However, the work distinguished between metric categories based on the stated (or in a few cases, presumed) intent of each study.

Having covered the general topics in the area of dynamic metrics (defined in the three classification schemes), we next provide a detailed explanation of the groups of metrics that are covered most often in the selected papers. There are several groups of dynamic metric topics that appear to be the central focus of the studies, described in the sections that follow. A detailed distribution of papers per topic and metric types is provided in Appendix B.

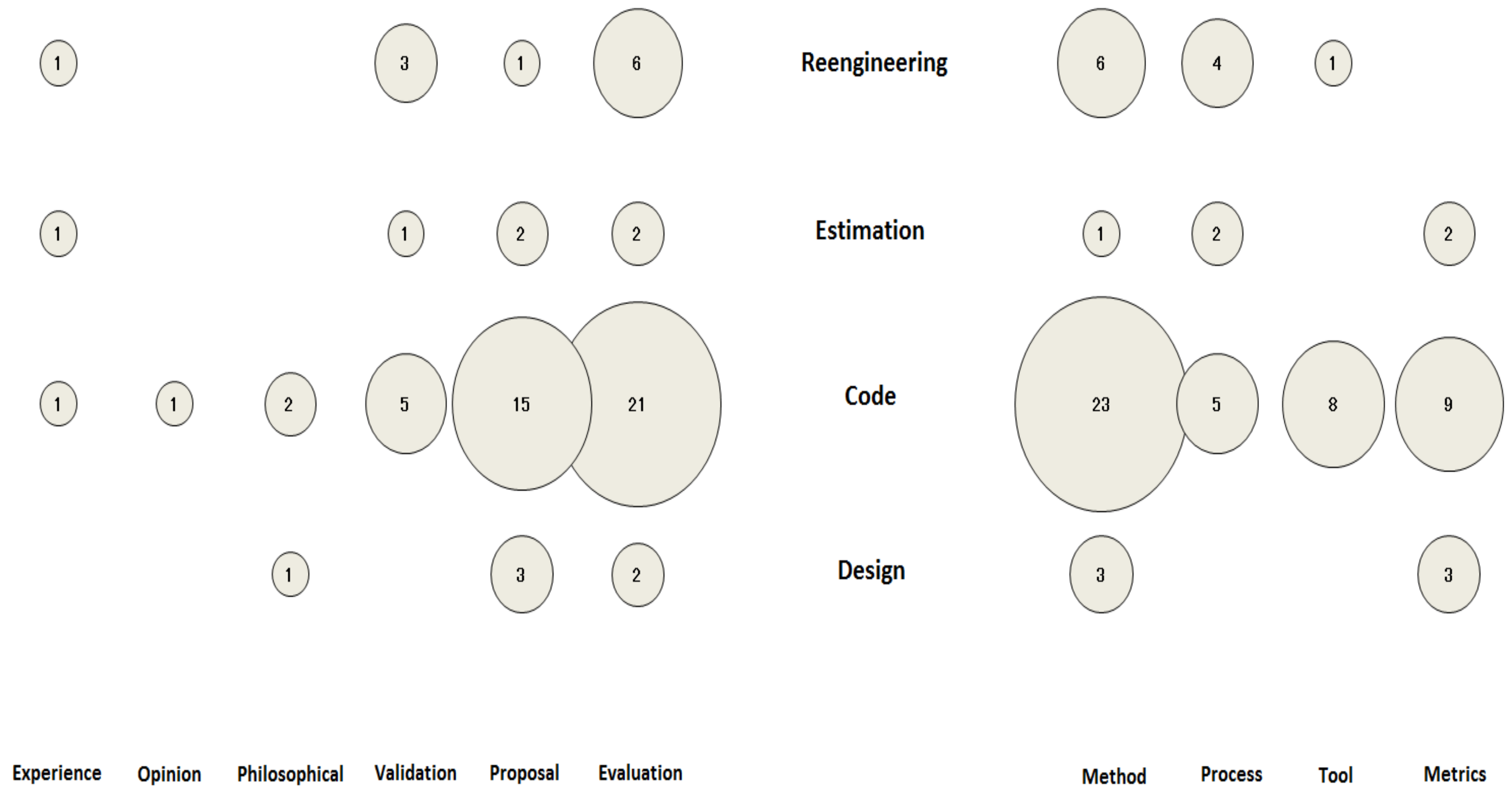


Figure 8. Map of research focus over research and contribution types

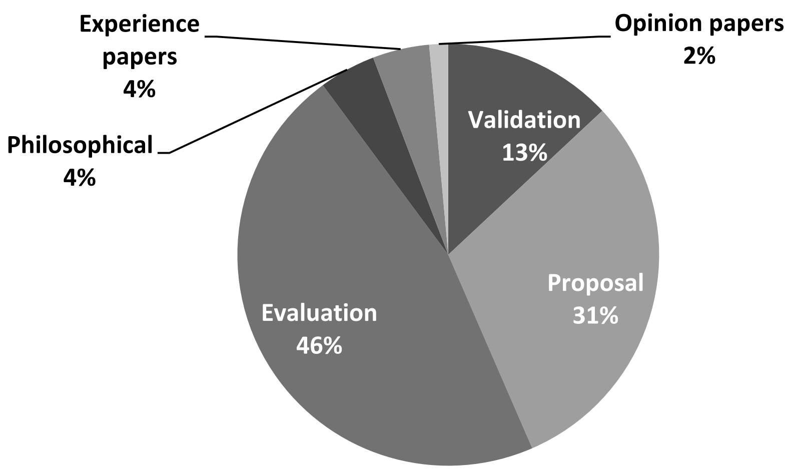


Figure 9. Articles distribution by research type

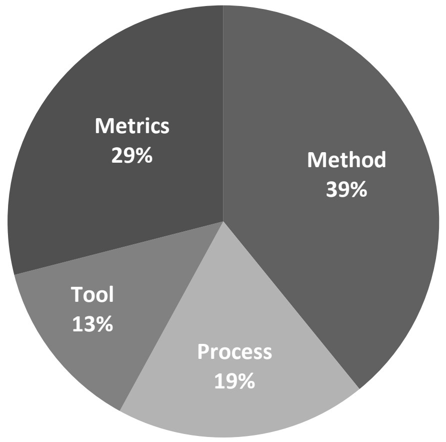


Figure 10. Articles distribution by contribution type

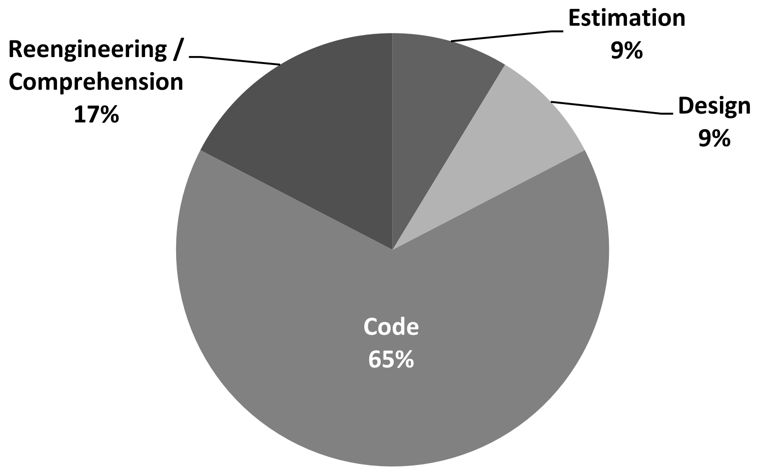


Figure 11. Articles distribution by research focus

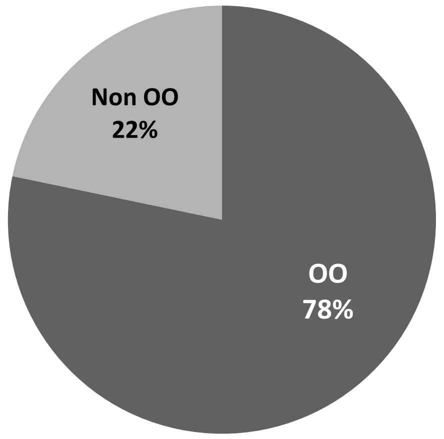


Figure 12. Articles distribution by programming focus

4.4.1 Coupling

A relatively strong body of research related to dynamic coupling was found (e.g. (Arisholm et al., 2004, Hassoun et al., 2005, Mitchell and Power, 2006, Yacoub et al., 1999)). Of note is that most of the dynamic coupling measurement works were motivated by the C&K metrics suite (Chidamber and Kemerer, 1994) and their well-known static coupling measure Coupling Between Objects (CBO). Cho et al. (1998) introduced a metric that assesses dynamic coupling at an object level by measuring the message passing load. Yacoub et al. (1999) proposed two dynamic coupling metrics (i.e. *Import* and *Export* object coupling) to measure coupling at the design level using Real-time Object Oriented Modelling (ROOM) charts. These authors later applied the same set of metrics to estimate and assess reliability risks during early phases of development (Yacoub and Ammar, 2002). Arisholm et al. (2004) introduced a set of code-level dynamic coupling metrics based on the dynamic analysis of systems. The authors found that dynamic coupling measures can be a good indicator of the complexity and change-proneness of a system. Burrows et al. (2010, 2011) empirically examined several dynamic coupling metrics, in the context of Aspect Oriented Programming (AOP). It was found that most of the existing AOP coupling metrics did not correlate well with several faults related specifically to aspect-orientation. Out of these metrics, the authors found that Base-Aspect Coupling (BAC) and Crosscutting Degree of an Aspect (CDA) were the two metrics that displayed the strongest correlation with faults (Burrows et al., 2010). Furthermore, the authors indicated that the extensions of C&K object-oriented based metrics had not proven to be good indicators of fault-proneness in AOP.

4.4.2 Cohesion

Cohesion is another reasonably well-studied topic and, as with coupling, most of the proposed dynamic cohesion metrics are based on the C&K metrics suite (Chidamber and Kemerer, 1994). One of the earlier works on runtime cohesion (Gupta and Rao,

2001) uses a novel program execution-based approach to measure the module (functional) cohesion of legacy systems, by applying a dynamic slicing approach to overcome the limitations of over-estimation resulting from the classic static slice. The more recent work of Gupta and Chhabra (2011) defined a set of dynamic metrics to measure cohesion at an object level. The authors defined four types of metrics to measure four levels of relationships. Their empirical evaluation showed that these new measures were more accurate when compared to other existing cohesion metrics.

A runtime form of C&K's *Lack of Cohesion in Methods* (LCOM) metric was introduced by Mitchell and Power (2004). Two new metric variants are the *Run-time Simple LCOM*, which is derived directly from the C&K static LCOM metric, and the *Run-time Call-Weighted LCOM* metric, which measures each instance variable by the number of times it is accessed at runtime. Three dynamic measures were proposed by Khurana and Kaur (2009) based on the Read/Write interactions between methods. These metrics were also inherited from the C&K static cohesion metrics. Cho et al. (1998) measured cohesion based on the message passing load, taking into account both the number of messages as well as the load carried in each.

4.4.3 Complexity

Like software quality, complexity is an amorphous concept that, when measured, is operationalized in other terms. That said, the studies considered here utilized the term 'complexity' so we have retained it for this discussion. Munson and Khoshgoftaar (1992) defined their *Functional Complexity* metric, said to measure the dynamic complexity of systems. This metric was further used in a later, related study (Munson and Hall, 1996) to estimate and examine the test effectiveness of software programs. Two additional dynamic complexity measures were introduced in their work, namely the *Fractional* and *Operational Complexity* metrics. The latter study (Munson and Hall, 1996) found a direct relationship between these dynamic metrics and software faults. Yacoub et al. (1999) later used the *Operational Complexity* metric to measure the 'dynamic complexity' of software. This metric is based on McCabe's static *Cyclomatic Complexity* metric and can be collected during the early stages of development using *State Charts* of the ROOM simulation modelling environment.

Voas (1992) introduced the *Revealing Ability* dynamic metric, which proposed to measure semantic software complexity by predicting a program's ability to allow faults to be undetected during dynamic random testing. Another run-time complexity metric was introduced by Mathur et al. (2010) based on decision points in code, where one option is chosen from an available selection.

4.4.4 Other Metrics

Burrows et al. (2011) used dynamic metrics to measure code churn[33], which has been shown to have a direct effect on the incidence of faults. Cai (2008) proposed a set of dynamic metrics that was used to measure the modularization of software components during maintenance tasks. This was achieved by comparing different versions of a program. The metrics considered the differences between versions, in terms of modules added, deleted, and changed.

A requirement-based dynamic metric was proposed by Cleland-Huang et al. (2001) as being useful in predicting network communication loads. The authors contended that this metric could be applied during the early stages of development, using data collected from a system's requirement specification and defining a 'typical' usage scenario. Mendes et al. (2005) empirically tested a size-based dynamic metric to measure the features and functionalities of web-based applications. This metric was used alongside other static metrics for the purpose of cost estimation during the early stages of development.

Gani et al. (2006) proposed a solution for dynamic metric collection to support adaptation via object mobility, for mobile applications. Six different metrics were used in their work to measure aspects related to execution frequency, performance,

[33] Code churn is the measure of the amount of added or modified code in a software component over time.

execution time, and resource utilization. In a similar work, Shtern et al. (2014) proposed a dynamic metric for cloud computing that is said to assess how effectively an application uses cloud infrastructure.

4.4.5 Metrics Suites

Several works have introduced metrics suites (i.e., sets of dynamic metrics) that could be used to collectively measure various aspects of software programs. These metrics could be collected either separately or as a set. In addition to broad coverage a key benefit of using a metrics suite is that there is potential for multiple measures of the same underlying construct (Chidamber and Kemerer, 1994), potentially of use for result triangulation.

Dufour et al. (2003a) proposed a set of dynamic metrics for *Java* programs that could be used to measure several runtime properties of software programs. These dynamic metrics were gathered into five main groups: size and structure of programs, data structures, polymorphism, memory, and concurrency. These metrics were examined empirically against several well-known *Java* benchmarks. The authors contend that these metrics could be used to capture relevant qualities; especially for compiler optimisation developers. More recently, Sarimbekov et al. (2013) proposed a similar set of dynamic metrics targeting JVM languages.

A metrics suite for *Component Based Development* (*CBD*) was presented by (Narasimhan and Hendradjaya, 2007). Several metrics, both static and dynamic, were designed to measure the complexity and criticality of component assembly. Röthlisberger (2010) designed and implemented five different dynamic metrics to enhance the *Eclipse IDE* analysis of Java applications, in order to help developers achieve a better understanding of their software. These metrics collect method execution and memory-related data.

4.4.6 Tools

Several tools have been implemented to help automate the dynamic measurement process. Some of these tools are implementations of some of the metrics suites just described. A summary of these tools is shown in Table 14.

Table 14. Summary of current dynamic metrics tools

Tool	Description
***J** (Dufour et al., 2003b)	*J is a tool designed to ease the dynamic metrics data collection process. The tool is used to gather, compute and present dynamic metrics data for Java programs.
A new debugging tool (Aggarwal et al., 2003)	A debugging tool to compute the number of executions for individual methods. This is the only tool found that has been designed specifically for the C language.
AOP Hidden-Metrics (Cazzola and Marchetto, 2008)	An AOP-based adaptable tool that collects dynamic metric data in a non-invasive way. They employ an AOP technique using AspectJ.
Senseo (Rothlisberger et al., 2009)	A plugin to enhance the traditional static information provided by Eclipse with various dynamic metrics information. *Senseo* collects both runtime information and performance-related metrics.
DynaRIA (Amalfitano et al., 2010)	A tool designed to support and enhance the comprehension of *Ajax* applications for the purpose of maintenance, reverse engineering and testing.
A new (CCRCs) profiler (Moret et al., 2010)	A profiler that uses (CCRCs) visualisation charts to enable efficient construction and navigation of large *Calling Context Trees* during execution. It also provides a visualisation environment for the collected dynamic data.

The *J tool (Dufour et al., 2003b) was one of the first tools designed to collect dynamic metrics' data. This tool supported the metrics suite designed in Dufour et al.

(2003a) (described in Section 4.3.5) and so collected measures of Size, Data structures, Polymorphism, Memory allocation and Concurrency and Synchronization. *AOPHiddenMetrics* (Cazzola and Marchetto, 2008) uses *AspectJ* to collect dynamic metrics in execution time. The data collected addresses *Coupling*, *Cohesion*, *Code Coverage*, *Code Execution*, *Memory Usage*, and *Concurrency* metrics. Rothlisberger et al. (2009) implemented an Eclipse plugin called *Senseo* that could be support to ease the measurement and analysis process, using the same set of dynamic metrics that was presented in (Rothlisberger, 2010). Similar to the tool developed by (Cazzola and Marchetto, 2008), *Senseo* also employs an AOP technique to gather the required runtime data. The main goal of *Senseo* is to dynamically analyse software and augment the static perspectives of *Eclipse* with dynamic metrics data, including method invocations, method execution, and counts of objects created during a particular execution scenario. These dynamic metrics are aggregated over several runs of the subject system, and the developer then decides which runs to take into account. The *DynaRIA* tool (Amalfitano et al., 2010) was designed to support the comprehension of *Ajax* web applications. In addition to other dynamic information, the tool collects dynamic metrics data such as size and code coverage metrics. Moret et al. (2010) presented a profiler that used visualisation charts, called Calling Context Ring Charts (CCRCs), to enable efficient construction and navigation of a large Calling Context Tree (CCT) while the program is being executed.

All of the works just described targeted the OO paradigm. Aggarwal et al. (2003) designed a tool to generate the execution sequence of modules during a run of any software. That said, this tool was mainly designed for procedural C programs, to measure the number of executions of all modules, using the *Most Frequently Executed Module* (MFEM) dynamic metric.

4.5 Discussion

The results of this mapping study indicate that issues related to dynamic metrics are receiving increased attention from researchers. Over half the body of relevant papers were published between 2007 and 2011. However, there has been a drop in the number of publication between 2012 and 2014, compared to the number of articles published between 2009 and 2011. In this section we address the research questions noted in Section 4.2.1.

RQ$_{4.1}$: Which aspects of dynamic metrics have been most frequently subjected to study?

The most widely studied aspects of dynamic metrics have been software complexity, memory allocation and usage, and code execution metrics. A relatively large proportion of research has been focused on coupling, cohesion and other complexity and maintainability metrics. Coupling has been the most studied single metric type. A number of studies have proposed new or amended sets of coupling-related metrics. Several empirical studies used dynamic coupling metrics to collect data for the purpose of software comprehension and/or reengineering.

RQ$_{4.2}$: Which aspects of dynamic metrics could be recommended as topics for future research?

In respect to the many dimensions of software quality, it is clear that complexity- and maintainability-oriented dynamic metrics have been the most widely discussed in the literature; however the same high level of attention has not been directed to metrics for other quality dimensions such as reusability, testability and portability. In our view dynamic metrics could be well suited to measure, and predict, testability. Measuring testability dynamically could be effective, particularly when considering different levels of testing (e.g., unit, integration, system) and the relationships between components. In addition, we considered the use of dynamic metrics at various levels of development and found that the proportion of investigations into design-level metrics is relatively low when compared to that for code-level studies. If useful metrics can be determined at the design stage this could help to reduce or minimize the risk of later costly errors and failures.

4.6 Threats to Validity

The main validity threat to this review study is the incomplete or inappropriate selection of publications. In spite of us following a systematic approach, it is still possible that we have missed some relevant studies especially if they were published in sources other than those we considered or that had not been cited in any of the articles selected in our search. To mitigate this risk we defined our search string for the automatic search alongside the search strategy. We also conducted our search using multiple automatic search sources. A manual search was then conducted to manually check for articles in a list of selected journals and conferences. In addition, we used a reference checking procedure to carefully look at the list of references in all identified articles. Thus, while it is still possible that we may have missed a small number of relevant papers, we would contend that the impact of such an oversight on the overall conclusions of the mapping study, given the number of papers that were selected and reviewed, would not be significant.

There is also a chance that some related papers have used terms other than those we used in our search string. If terms other than "dynamic metrics", "dynamic measurement", or "runtime metrics" were used then the possibility of us missing a study is high. To avoid such a problem we repeatedly refined our search string and performed sequential testing in order to recognize and include as many relevant studies as possible. In addition, we conducted reference checks on all reference lists of the selected articles on the topic, to locate any missing influential articles. The selected studies were then examined and subsequently added to the final set of papers to be reviewed. In our view this was of use in limiting the number of missing influential articles (although we are unable to 'prove' this). Furthermore, the manual search we conducted was intended to fill any gaps by directly targeting relevant reputable publishing venues.

Another possible threat to the validity is the way the studies are classified. The classification was done mainly by one researcher. It is possible that some of the articles can be classified differently by different researchers due to the possibility that some of these articles can fit into multiple categories of the same classification scheme.

4.7 Summary

This Chapter has reported a systematic mapping study of dynamic software metrics. In summary, this work presented 1) a general overview of the field of dynamic software metrics, 2) the selection of key works in the area based on number of inclusion and exclusion criteria, and 3) a mapping and classification of the selected articles. Detailed review results are shown in Figure 6, and a map of all the selected articles is shown in Figure 8. The classification results of these works are given in Figures 9-12. A list of all recognized metrics is provided in Appendix B.

The results of this mapping study indicate that there is indeed ongoing interest in dynamic metrics among the software engineering research community. That said, most prior studies on dynamic metrics have focused specifically on software complexity aspects (either directly or indirectly). Moreover, with most of the studies focusing on OO systems, a great deal of emphasis has been directed towards OO-related factors such as coupling and cohesion. Beyond complexity, these metrics have been identified as being relevant to a number of other software quality attributes, such as complexity and maintainability. Given this, the empirical analyses that follow investigate the potential of these and other relevant metrics in relation to testability and test quality.

The following chapter presented the results of our first experiment on the use of visualisation and dynamic analysis to explore the distribution of unit test suites.

Chapter 5 Exploring the Distribution of Unit Test Suites

5.1 Introduction

This chapter analyses test adequacy in several OSS. It presents a novel approach that combines dynamic coupling and unit test static data in order to enable developers and those undertaking re-engineering to gain greater visibility into unit tests' distribution (i.e., the distribution of the unit tests over a system's classes), across five OSS. In this experiment, a visual representation (based on complex network and dependency graph theories) of dynamic information is developed to present the dynamic information directly in relation to unit tests.

The field of software visualisation has long offered substantial promise in aiding software developers and maintainers to better understand certain aspects of software behaviour (Maletic et al., 2002). Researchers have also suggested that software metrics should be examined through appropriate visualisations, thus achieving improved understanding beyond the 'raw' numbers of the metrics alone (Lanza and Marinescu, 2006). In short, using visualisations to support program comprehension and the understanding of software artefacts (including test artefacts) appears to be effective and useful (Lange and Nakamura, 1997, Jerding and Rugaber, 2000, Cornelissen et al., 2007, van Rompaey and Demeyer, 2008, Cornelissen et al., 2011). Section 2.4.1 provides a discussion on the previous works that used test visualisation as a means of supporting developer comprehension and understanding.

We contend that visualising such data could be especially helpful when maintenance and reengineering activities take place, as the visualisation process elucidates the hierarchy of the production classes and the distribution of unit tests corresponding to the production classes. In addition, such visualisations would provide developers and maintainers with a high-level view of the dependencies between different classes within a system and the possible utilisation of methods for future testing activities, i.e., what components are not being tested, the degree to which other components are tested, and where testing effort should be focused.

The experimental design is discussed in the following section.

5.2 Experimental Design

Understanding test code and its use in systems is an important task in software development, and particularly during software maintenance, reverse engineering and refactoring. In the object-oriented paradigm, production code and test code are similar in nature (i.e., they are written in a similar manner); thus, analysing and understanding test code requires similar skills and methods as used with production code. The following subsections present the specific objectives and contributions of this experiment, along with a description of the metrics used in its conduct.

5.2.1 Objectives

The main objective of this experiment is to explore unit test distribution by using a novel visualisation approach that combines specific static and dynamic information. We demonstrate the application of the visualisation on sample OSS, including systems of different sizes and with different test coverage levels.

In achieving the above objectives this research will enable us to assess whether dynamic information, here represented by dynamic coupling, might be useful when added to unit test information to represent the distribution of unit tests in OSS. The goal is to develop a visualisation approach that combines dynamic information associated with production code and test information, with a view to supporting better understanding of the distribution of unit test suites in OSS. Then this experiment studies the distribution of unit tests to determine whether production classes and unit tests are evenly distributed; that is, do all highly and/or tightly coupled classes have dedicated unit tests?

5.2.2 Contribution

The findings of this work contribute to the general body of knowledge on program comprehension and understanding (and more specifically, test comprehension) by enabling the visualisation of a new combination of static and dynamic data that could aid the test understanding process. The methods developed in this study should provide developers with knowledge of the unit test distribution in OSS.

One possible use of the proposed visualisation is when maintenance and reengineering activities are planned. The visualisation should enable software developers, maintainers and reengineers to explore the distribution of unit tests in relation to the dynamic behaviour of the software before conducting their work. It should also benefit program understanding by providing a visual representation of the dependencies based on actual use of the functional capabilities of the system. Newcomers to a project could also use the proposed visualisation to understand which aspects have been directly covered with unit tests in relation to their dynamic dependencies view (van Rompaey and Demeyer, 2008). The visualisation could also be beneficial in dynamic or Agile-like development contexts, in which unit tests serve as a key form of documentation (Cornelissen et al., 2007).

5.2.3 Metrics Definition

In this study Dynamic Coupling has been selected as one of the system characteristics to measure and investigate regarding its relationship to class testability. Coupling has been shown in prior work to have a direct impact on the quality of software, being linked in particular to software complexity and maintainability (Offutt et al., 2008, Al Dallal, 2013, Arisholm et al., 2004). It has been shown that, all other things being equal, the greater the coupling level, the greater the complexity, and the harder it is to maintain a system (Chaumun et al., 2000). This suggests that it is reasonable to expect that coupling will be related to software testability in general. Dynamic rather than static coupling has been selected for this work to address some of the expected shortcomings of the traditional static measures of coupling. As noted above, coupling has generally been measured statically, based on limited structural properties of software. This misses the coupling that occurs between objects at execution-time – such as typical interactions that happen during polymorphism and dynamic binding. Dual consideration of this form of coupling should capture a more complete picture (as it captures *runtime* dependencies between different classes/objects (Arisholm et al., 2004)) and so relate better to class testability. The notion of measuring dynamic coupling is quite common in the emergent software engineering research literature.

For the purposes of this work, dynamic coupling metrics that capture coupling at the object level (at runtime) are used. In this work, we define dynamic coupling following the definition proposed by Zaidman and Demeyer (2008): dynamic coupling is defined based on an analysis of runtime interactions between classes/objects - "two objects are dynamically coupled when one object acts upon the other. Object X is said to act upon object Y, when there is evidence in the execution trace that there is a calling relationship between objects X and Y, originating from X. Furthermore, two classes are dynamically coupled if there is at least one instance of each class for which that [*sic*] they are dynamically coupled holds" (p 391).

There are various ways to measure dynamic coupling. The specific measure of coupling used here is based on runtime method invocations/calls: *two classes, class A and class B, are said to be coupled if a method from class A (caller) invokes a method from class B (callee) at run-time, or vice versa*. Multiple invocations to the same class are still counted as a single (*one*) coupling.

In this experiment the *Dynamic Coupling Between Objects (DCBO)* metric is used. As the name implies, DCBO is the dynamic form of the well-known CBO metric (Chidamber and Kemerer, 1994). For any class, the DCBO metric computes the total number of classes that are invoked by that class during program execution (and note that self-calls to cohesive methods from the same class are excluded). This metric is collected through an AOP approach using the *AspectJ* framework (see Section 3.4). The following shows our defined AOP (AspectJ) rules that detect calls made during typical program execution:

```
pointcut capture() : call (* *..*(..)) // capture all method calls made during the execution
pointcut exclude(): !call (system-under-examination)// excludes all calls that are not relevant to
the program under examination such as default java libraries and default AspectJ compiler's calls
before() : capture() && !excluded()
{
            // capture the name of the caller class
    caller = thisEnclosingJoinPointStaticPart.getSignature().getDeclaringTypeName();
            // capture the name of the callee class
    callee = thisJoinPoint.getSignature().getDeclaringTypeName();
}
```

As explained above, dynamic instead of static coupling is used to measure coupling between classes. As shown in Chapter 4 dynamic coupling has received increasing research attention in recent years, but looking at the relation between such metrics and testability has not been done previously.

5.3 Data Collection and Execution Scenarios

Five of the OSS from the list presented in Section 3.2.1 were used in this study: FindBugs, JabRef, Dependency Finder, MOEA and JDepend. General characteristics of these systems are shown in Table 4. As is evident in Table 4, the five selected systems represent a variety of sizes (one large, three medium and one small). Table 15 shows detailed test coverage information for all five systems.

In order to arrive at dynamic analysis values that are associated with typical, genuine use of a system, the selected execution scenarios must be representative of such use. The goal is to mimic 'actual' system behaviour, as this will enhance the utility of our results. Execution scenarios are therefore designed to use all key system features, based on the available documentation and user manuals for the selected systems, as well as our own prior knowledge of these systems. Note that all five systems have

GUI components, and the developed scenarios assume use via the available GUI. Details of the particular execution scenario developed for each system, and used in this experiment, now follow.

FindBugs: FindBugs' main GUI tool is used to analyse JAR and source code files of three other Java OSS, two of large size (FindBugs itself and Apache JMeter) and one of medium size (Dependency Finder). During execution, cloud-based storage was activated by loading the tool's external cloud plugin. Finally, all analysis reports were exported for all three systems in various formats.

JabRef: The tool is used to generate and store a list of references from an original research report. References of all types supported by the tool were included (e.g., journal articles, conference proceedings, reports, standards). Reports were then extracted using all available formats (including XML, SQL and CSV). Finally, the list of references was managed using all the provided features. All additional plugins provided at the tool's website were added and used during this execution.

Dependency Finder: This scenario involves using the tool to analyse the source code of three large systems (FindBugs, Apache JMeter, and Apache Ant) and one medium-sized system (Colossus). This scenario involves computing dependencies, dependency graphs and OO metrics at all layers (i.e., packages, classes, features). Analysis reports were extracted and stored individually in all possible formats.

MOEA: The tool has a GUI diagnostic tool that provides access to a set of algorithms, test problems and search operators supporting multi-objective optimization. The diagnostic tool was used to apply those algorithms on all of the predefined test problems. The algorithms were executed at least once on each problem. The tool was then used to display metrics and performance indicators for all results obtained from those different problems and algorithms. Statistical results of these multiple runs were displayed and stored at the end of the execution.

JDepend: A small GUI was designed as part of this research to provide access to all of the quality assessment and reporting functionalities of JDepend (Note: this additional code was excluded from the measurement collection and analysis.) The tool was then used to load and analyse four different OSS, three of medium size (Dependency Finder, JabRef, and barcode4j) and one of large size (FindBugs). All three user interfaces provided by the tool (namely: swing-graphical, textual and XML) were used during this execution.

Table 15. Test coverage data

System	Class Coverage	Statement Coverage	Branch Coverage	Line Coverage	Methods Coverage
FindBugs	26.5%	13.3%	6.9%	14.1%	18.6%
JabRef	46.7%	29.6%	14.2%	29.5%	31.0%
Dependency Finder	59.5%	59.8%	44.3%	57.2%	43.3%
MOEA	86.5%	77.2%	46.2%	66.6%	60.5%
JDepend	41.8%	25.9%	14.9%	28.1%	28.5%

5.4 Results

This section presents the results and analysis of the empirical investigation of the proposed measurement and visualisation approach. A dependency graph is used to visually depict the dependencies between classes with each system. Dependencies, shown here with undirected edges, represent method invocations (calls sent or received) between classes, shown as nodes. An undirected edge between nodes *A* and *B* means that the two nodes are coupled. That is, a dependency between classes *A* and *B* represents at least one invocation from a method in class *A* to a method in class *B*, and/or vice versa. A description of the dependency graph node symbols is provided in Table 16. For tightly coupled classes the size of the vertex represents the relative degree of coupling measured by DCBO.

Graph metrics are used here to quantify the level of association that a node (i.e., class) has with other nodes in the graph. Centrality, in graph theory, is defined as the level of reachability of two different nodes of a graph (Boccaletti et al., 2006). Graph centrality is a well-known concept in graph theory that has been applied increasingly in recent times to analyse *Complex Networks*[34]. It has been long used across multiple domains to analyse large, complex networks such as those used in Social Network Analysis (SNA)[35] (Freeman, 1978, Borgatti, 2005).

Table 16. Dependency graph node symbols

Symbol	Description
■	Tightly coupled class, with at least 1 associated unit test
▲	Tightly coupled class, with no associated unit test
◇	Loosely coupled class, with at least 1 associated unit test
●	A production class with no associated unit test

In particular, we identified two centrality metrics to be used: *Degree Centrality* and *Betweenness Centrality*. *Degree Centrality* is defined as the number of ties upon a node in a graph (Borgatti, 2005), and it is measured based on the total number of links (connections) for a node. This metric directly reflects the dynamic coupling information, which is obtained from the DCBO metric, reflecting messages sent or received by a class (also known as Import and Export Coupling). *Betweenness Centrality*, on the other hand, is defined as "the share of times that a node *i* needs a

[34] *Complex Networks* are widely used in many fields; include physics, biology, epidemiology and computer network and telecommunications.

[35] SNA is the study of relationships between different social entities (such as communications between different members of a social group) through the use network graphs.

node *k* (whose centrality is being measured) in order to reach a node *j* via the shortest path" (Borgatti, 2005). In other words, this metric calculates the number of times a node acts as a *bridge* between two other nodes in the graph. Both centrality metrics are computed for each individual node on the graph.

Figures 13 to 17 show dependency graphs for all five systems. Given that all visualisation graphs should be presented in a complete form and seen in clear colouring, a full-size, high-resolution version of these graphs is provided in an external webpage[36].

Table 17. Centrality metrics for JDepend

Class	Degree Centrality	Betweenness Centrality	Unit test	Class	Degree Centrality	Betweenness Centrality	Unit test
framework.JDepend	9	92.8	Yes	FileManager	2	0.5	Yes
JavaPackage	9	72.6	Yes	AbstractParser	2	18.0	No
swingui.JDepend	5	37.5	No	PropertyConfigur ator	2	0	Yes
JavaClass	5	8.6	No	xmlui.JDepend	2	0	No
PackageComparator	4	5.0	No	ParserListener	1	0	No
textui.JDepend	4	3.2	No	AfferentNode	1	0	No
JavaClassBuilder	3	34.0	Yes	DependTree	1	0	No
PackageNode	3	18.0	No	DependTreeModel	1	0	No
ClassFileParser	3	0.8	Yes	EfferentNode	1	0	No
PackageFilter	3	0.5	Yes	StatusPanel	1	0	No

As shown in Table 17 and visually in Figure 13, the *framework.JDepend* and *JavaPackage* (highlighted) classes of the JDepend system are shown to have the highest levels of (Degree and Betweenness) Centrality. Both classes are also directly

[36] http://goo.gl/nuGZ4u

tested through dedicated unit tests. Other classes, including *swingui.JDepend* and *JavaClass*, have high levels of Degree Centrality (both have the second-highest value) but have no associated unit tests. In contrast, we also note that the *FileManager* and *PropertyConfigurator* classes have dedicated unit tests associated with them even though they are not shown to be central to the system's operation (i.e., their Centrality levels are low, especially in terms of Betweenness Centrality).

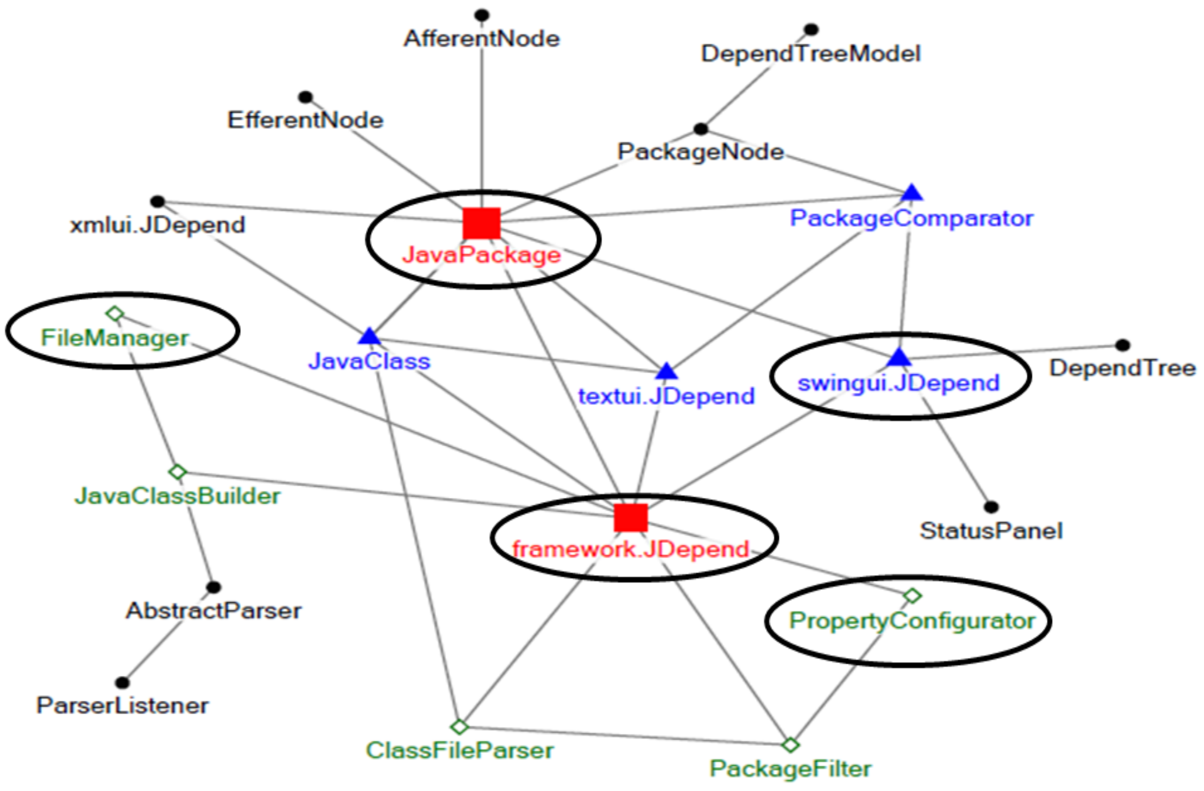


Figure 13. JDepend full dependency graph

Table 18 shows a comparison of the Centrality values (both Degree and Betweenness Centrality) for tested classes across the five systems, using a proportion of classes from the 'top' and 'bottom' of their ranked lists. For each system, Centrality values are ranked and then divided into four groups based on three *quartile* data points. The 1st (Q1 - the lower) and the 3rd (Q3 - the upper) quartiles split off the bottom and top 25% of the data points in terms of their centrality values, respectively, whereas the 2nd quartile (Q2 - the median) reflects the middle 50% of the data. Those classes with Centrality values above the Q3 threshold are relatively highly coupled, and those with values below the Q1 threshold are coupled loosely.

Table 18. Levels of centrality in all examined systems

System	Number and proportion of tested classes above Q3 for Degree Centrality	Number and proportion of tested classes above Q3 for Betweenness Centrality	Number and proportion of tested classes below Q1 for Degree Centrality	Number and proportion of tested classes below Q1 for Betweenness Centrality
FindBugs	11 7%	4 2%	1 1%	2 1%
JabRef	9 13%	11 15%	9 13%	4 6%
Dependency Finder	31 69%	24 53%	3 7%	19 42%
MOEA	25 66%	21 55%	17 45%	20 53%
JDepend	2 40%	3 60%	0 0%	2 40%

For the Dependency Finder system (Figure 14), classes with Centrality values above the Q3 threshold are examined (being 46 classes, with some exceeding the threshold for both Centrality measures). It is found that almost half of the classes in the Q3 threshold had no associated unit tests. For example, the *dependency.Printer* (highlighted) class has a Degree Centrality value of 53 (which is the second highest value in the system) and its Betweenness Centrality is 2032 (ranked fifth highest in the system), yet it has no associated (dedicated) unit tests. The same applies to *dependency.VisitorBase* (highlighted), which has a Degree Centrality of 52 (third largest Degree Centrality value) and has a Betweenness Centrality of 1283, and yet has no associated unit tests. These classes are considered to be central to the system based on its dynamic coupling values. In contrast, we observed other classes with very low levels of Centrality but with dedicated unit tests. For example, the *RegularExpressionParser* and *PrinterBuffer* classes both have devoted unit tests even though they have the lowest Centrality values, with only a value of one for Degree Centrality and zero for Betweenness Centrality. This latter result indicates that these classes do not appear to be central to the system's operation in terms of their dynamic coupling. A similar pattern is repeated in JabRef. As shown in Figure 15, the central classes of *Globals*, *JabRefPreferences* and *BasePanel* have no devoted unit tests while they still have high Degree and Betweenness Centrality values (these classes have the highest two Degree Centrality values respectively). On the other hand, classes such as *CaseChanger* and *DOICheck* appear to be less central (i.e. with low Centrality values) but they still have dedicated unit tests. In considering the Q3 classes by Degree Centrality in JabRef (71 classes), only nine classes (among the 71) were found to have dedicated unit tests. Similarly, only eleven classes with the highest Betweenness Centrality measure (among those 71 classes, which form only 15% of the classes in Q3 threshold) were found to have dedicated unit tests.

A generally similar pattern of unit tests' distribution is evident in all other examined systems. Figures 16 through 17 show dependency graphs for MOEA and FindBugs, respectively. Full *Centrality* metrics values (i.e., Degree and Betweenness Centrality) for all system are provided in the Appendix C.

In regard to the MOEA system (Figure 16), unit tests are present for 25 (66%) of the classes above the Q3 Degree Centrality threshold and for 21 (55%) classes above the Q3 value for Betweenness Centrality. However, MOEA also has the highest proportions of tested classes below the Q1 Centrality measure thresholds of the five systems considered, with 45% and 53% of these classes having unit tests. This may be a reflection of the generally high levels of test coverage in MOEA, as MOEA has around 87% class coverage (which means that 87% of the production classes are covered by unit tests, see Table 18). The lowest percentages of tested classes above Q3 for both Degree and Betweenness Centrality are evident for FindBugs (although it is also the largest of the five systems examined). It has 11 (~7%) classes with associated unit tests among the 164 classes in Q3, and only 1 tested class (< 1%) in Q1 for the Degree Centrality classes. For Betweenness Centrality, there are 4 tested classes in Q3 and 2 tested classes in Q1.

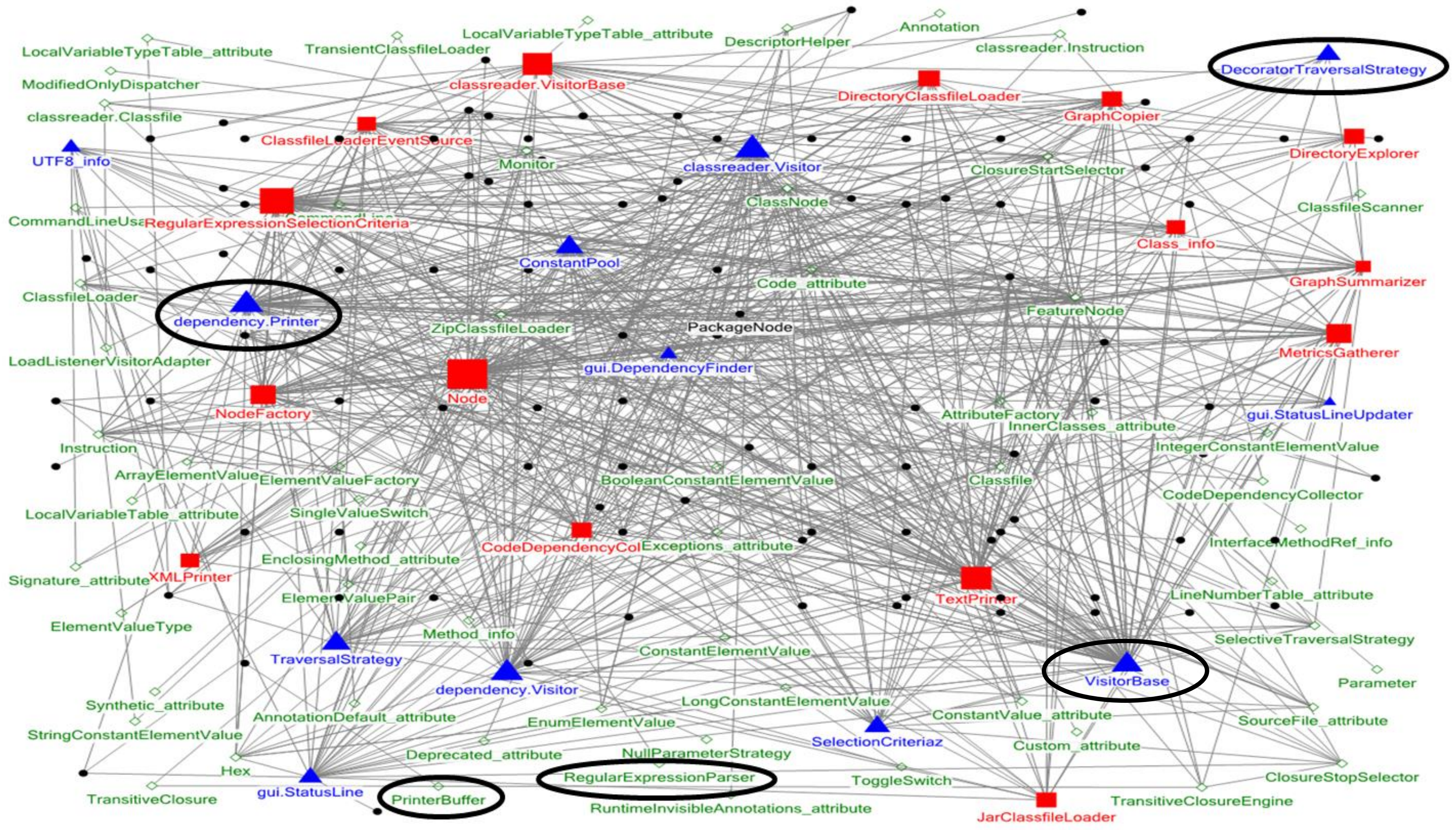


Figure 14. Dependency Finder dependency graph

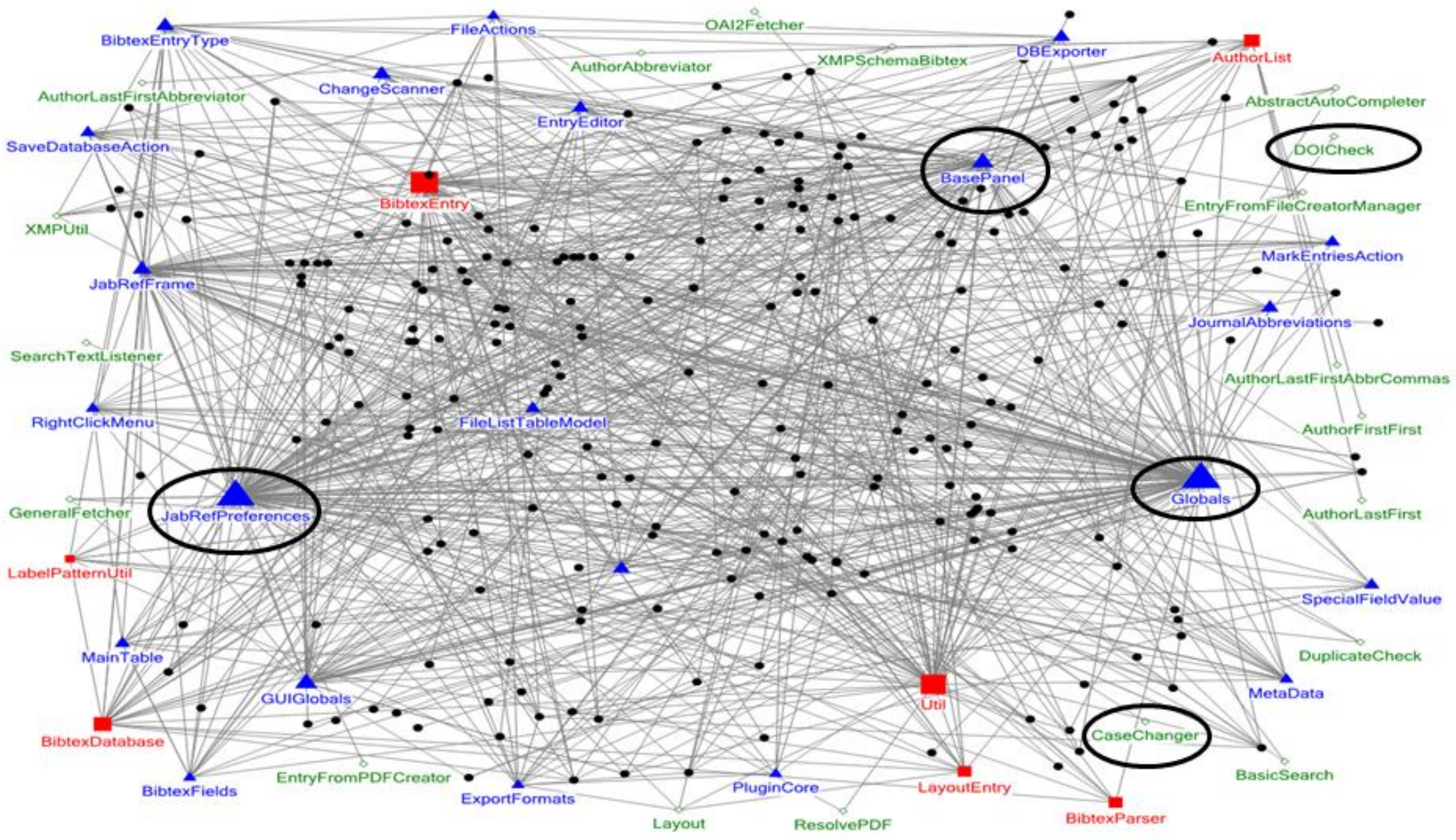


Figure 15. JabRef dependency graph

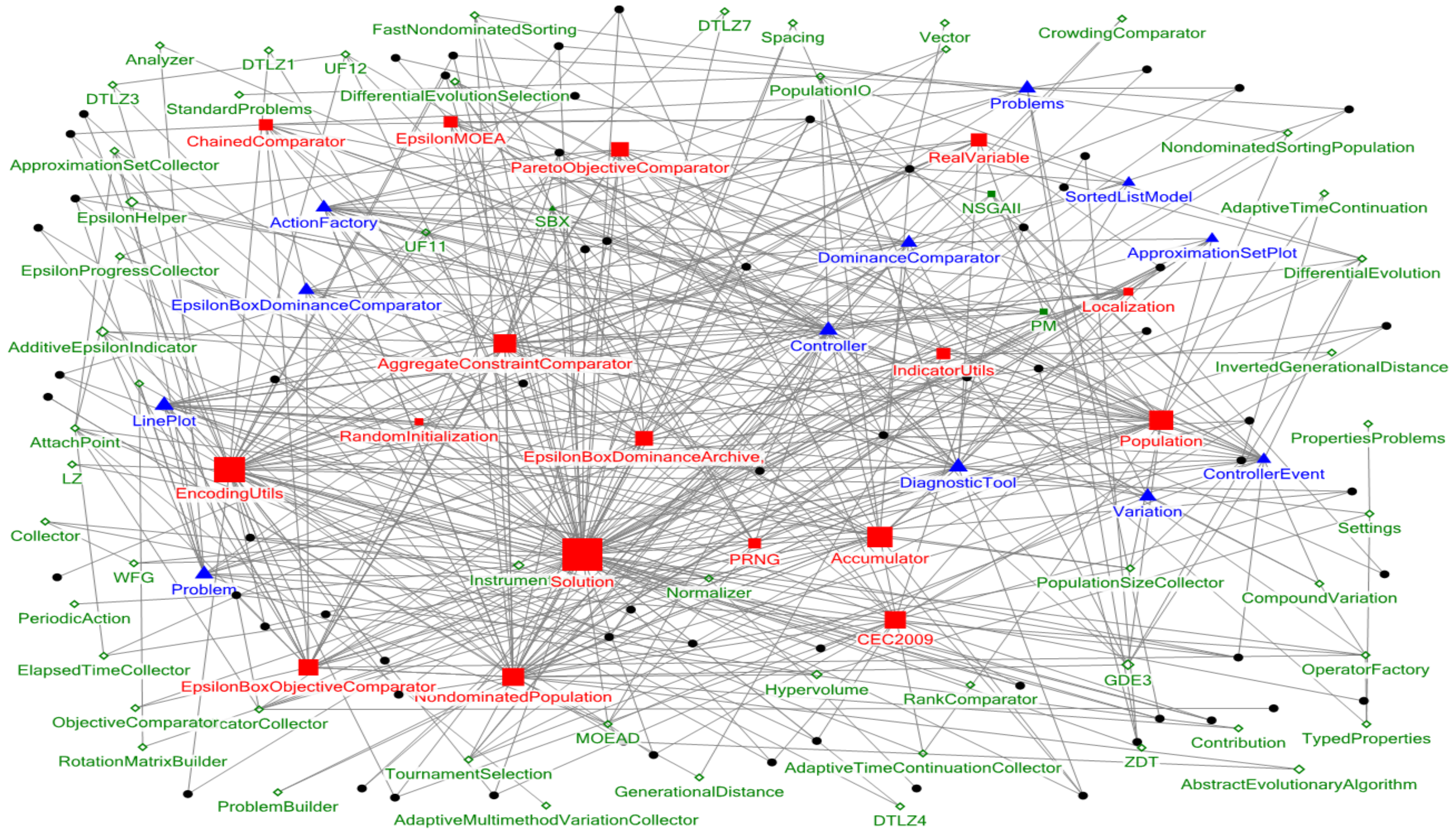


Figure 16. MOEA dependency graph

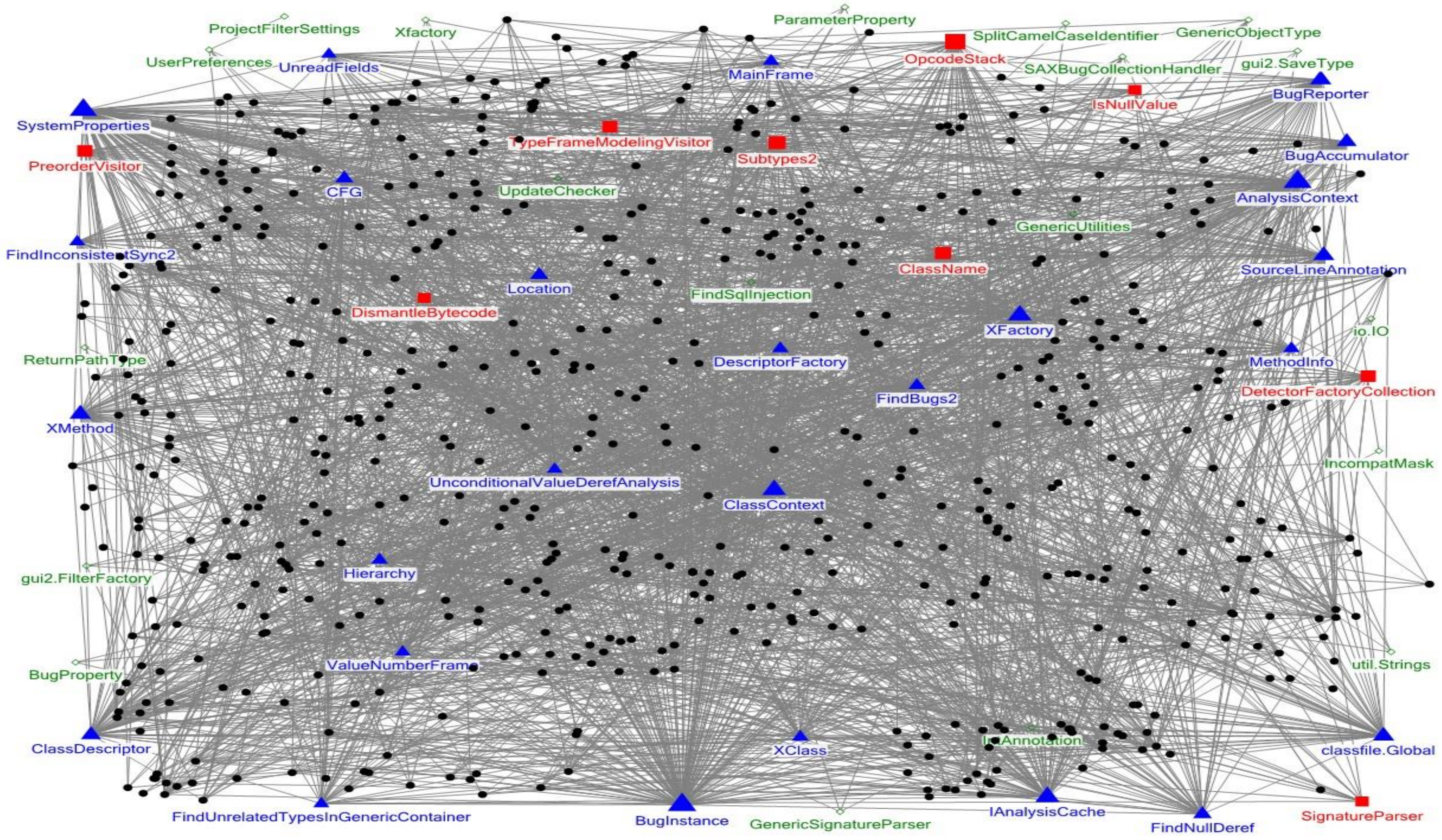


Figure 17. FindBugs dependency graph

To provide a more formal statistical analysis of the relationship between the two Centrality metrics' values and the availability of unit tests for production classes the non-parametric (two-tailed) *Mann-Whitney U test* is used (as the data come from non-normal distributions – see Section 3.2.2 for more details about this statistical test). The following research hypothesis is investigated: *"there is a significant difference between the Centrality metrics' values of production classes with associated unit tests and those without associated unit tests"*. In addition, the effect size (ez) of these differences is calculated using equation (2) in section 3.2.2 and is then classified into small, medium and large using Cohen's classification (Section 3.2.2).

Table 19.Centrality metrics Mann-Whitney U test results with effect size

Metrics		**FindBugs**	**JabRef**	**Dependency Finder**	**MOEA**	**JDepend**
Betweenness Centrality	p	0.00*	0.50	0.00*	0.08	0.11
	ez	0.12	0.04	0.27	0.14	0.36
Degree Centrality	p	0.01*	0.16	0.03*	0.45	0.26
	ez	0.10	0.08	0.16	0.06	0.25

Table 19 reports the results of the Mann- Whitney U test (with effect size) of the Centrality metrics and the presence of unit tests. All Significant p-values (p) are marked with an asterisk (*). As shown in Table 19, significant p-values are shown for only two of the five systems examined (i.e., FindBugs and Dependency Finder). Furthermore, even though the p-values are significant in these systems, the effect size values are small for both Centrality metrics. No significant values are shown for the other three systems. This result leads to a rejection of the hypothesis – there is no evidence of a significant difference between Centrality metrics' values of production classes with associated unit tests and those without associated unit tests.

5.5 Discussion

Several observations can be made based on the results just presented. The main observation, enabled by the visual representations of the dependency graphs and the two graph *centrality* metrics, is that there is no statistically significant association between the *centrality* metrics of Degree Centrality (which is also a representation of the dynamic coupling level in the systems) and Betweenness Centrality, and unit test coverage – unit tests do not appear to be distributed in line with the systems' dynamic coupling and centrality values. In all five OSS examined, it is evident that many classes (i.e., over 40% of the classes as shown in three of the five examined systems) that are loosely coupled and have few connections have received testing attention and effort, as they have dedicated unit tests. Loosely coupled classes, shown at the outside of the graphs, have fewer connections and so are not intensively accessed by other classes. On the other hand, high proportions of classes in each system (up to 69% of the classes as shown in Dependency Finder) that are tightly coupled, in terms of being linked to or accessed by other classes, have no dedicated unit tests.

Of particular note is that this distribution pattern is present in all five systems, regardless of their other test coverage information, such as class and statement coverage (although, the specific numbers of tested and untested classes naturally varies from one system to another). As shown in Table 19, it is evident that the proportion of unit tests in relation to coupling and centrality levels is different in all five systems. This suggests that, even in mature OSS such as those used in this experiment, the dispersed nature of contributions to the project may mean that test distribution can be uneven, and provision of tests is reliant on the commitment or otherwise of individual developers.

We acknowledge that the current visualisation tends to become dense with large systems, i.e., when the numbers of nodes and edges increase (as in the case of FindBugs – see Figure 17). In such cases the visualisation could be simplified if needed, by reducing the numbers of nodes and edges that are depicted in the graph. One option could be to show only certain classes (i.e., nodes) of interest and to hide all other classes. For instance, an interest in tested classes would mean that only classes with associated unit tests would appear in the visualisation – Figure 18 shows such a simplified version of FindBugs (after removing additional nodes and edges). The results presented in this experiment suggest that the distribution of unit tests may require more attention from software engineers and testers, but also from those managing software development. We contend that the suggested visualisation can help in focusing and optimising testing effort by allowing engineers to identify and initially target central system classes and to dedicate relatively less effort to non-central classes. We also suggest that the centrality metrics themselves could be helpful in providing quantitative support for the visualisations of the dependency graphs. The two centrality metrics provided us with a comprehensive insight into the levels of dependency between system classes. Such data could be used to supplement other test optimisation and prioritization techniques, alongside other considerations, to enhance future testing decisions. The same approach could also be applied to explore test distribution at smaller scales, such as at sub-system level or even at a package level.

While we believe that the results we have achieved so far are interesting, we readily acknowledge the need for user evaluation of the utility of the visual representations. The work presented in this chapter is exploratory (a proof of concept) in nature, and it was driven purely by an open question as to the feasibility of combining dynamic and static metrics with visualisations in the context of testing. Certainly we recognise the need for external evaluation, and this is planned for future work. This could be undertaken through a controlled experiment using real software developers/testers.

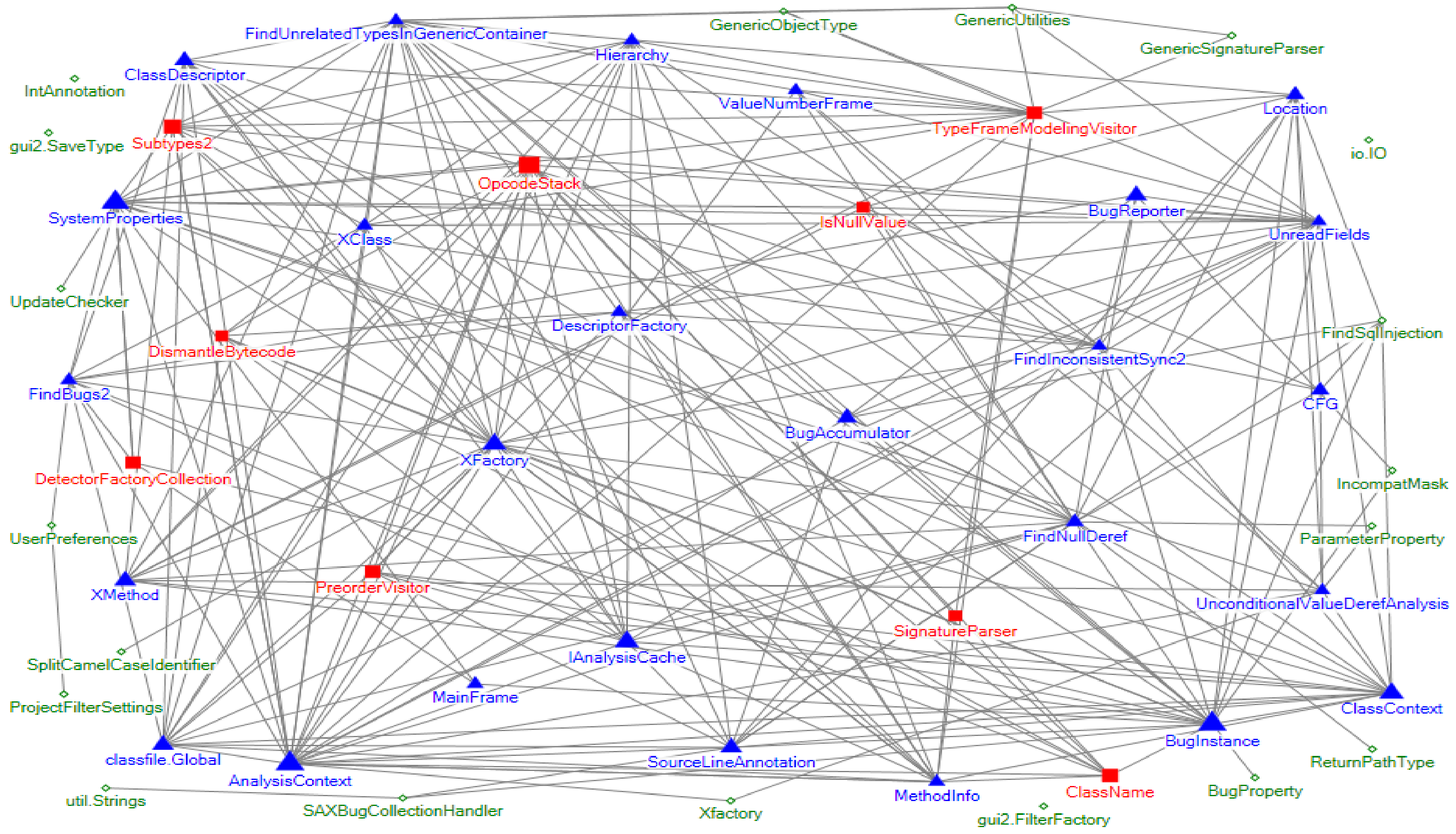


Figure 18. FindBugs dependency graph.

5.6 Threats to Validity

A number of threats to the validity of the results presented above are acknowledged here. Note that the nature of some of these threats is explained more fully in Section 3.5.

Selection of the Execution Scenarios

The selection of the execution scenarios is another possible threat to the validity of our results. Execution scenarios are designed to mimic as closely as possible 'actual' system behaviour, based on the available system documentation and, in particular, indications of each system's key features. However, it is acknowledged that the selected scenarios might not be fully representative of the typical uses of the systems. Analysing data that is collected based on different scenarios might give different results. This is a very common threat in most dynamic analysis research. However, we tried to mitigate this threat by carefully checking user manuals and other documentation of each of the examined systems and deriving the chosen scenarios from these sources. Most listed features were visited (at least once) during the execution. More scenarios will be considered in the future in order to extend the presented analysis and to compare the results obtained from these different scenarios.

Generalisation of Findings

Results discussed here are derived from the analysis of five OSS (including one large, three medium and one small system). This threat is also discussed in more detailed in Section 3.5.2.

Availability of Testing Information

The varied availability of detailed testing information could be another threat to the validity of this experiment. Whatever test information was available for the five systems was used in the analysis, but this did not extend to information regarding the testing strategy employed. Test strategy and criteria information could be informative if combined with the test metrics, given that test criteria can inform testing decisions, and the number of test cases designed is highly influenced by the selected test strategy. Moreover, a more comprehensive picture of the analysis could be gained by also considering indirect tests.

Test Quality

Finally, no attempts were made to direct attention to test quality – the intent at this stage was to investigate the existence or otherwise of unit tests for system classes. An analysis approach that considers both the quantity *and* quality of the tests developed would seem likely to be optimal, however, and should be the subject of future research.

5.7 Summary

This chapter has presented a new visualisation approach that combines dynamic information obtained from production code with static test information to depict the distribution of unit tests in OSS. Five such systems of different sizes were selected for examination in this experiment. For each system, a full dependency graph was generated to show the dependencies between classes using the collected dynamic coupling information. Test information was then extracted and added to the dependency graphs to illustrate how unit tests were distributed in comparison to the dynamic coupling information. The goal of this visualisation is to assist engineers and maintainers – and their managers – to observe and understand the distribution of unit tests in a software system based on a dynamic view of that system. The visualisation is further supported by the use and analysis of graph Centrality metrics that provide insight into the relationship between production classes and their unit tests.

Having investigated unit test distribution based on dynamic analysis in this chapter, the following chapter explores the relationship between particular dynamic software properties (i.e., runtime characteristics) and class testability. The goal is to provide a general understanding of what affects class testability and how dynamic analysis can help in this regard.

Chapter 6 Investigating Class Testability through Dynamic Analysis

6.1 Introduction

The diversity of design and code characteristics that can affect the testability of a software product has been the subject of a large body of work. For example, the relationships between internal class properties in OO systems and characteristics of the corresponding unit tests have been investigated in several previous studies (e.g., (Bruntink and van Deursen, 2006, Badri et al., 2011, Zhou et al., 2012)). In these studies, OO design metrics (drawn mainly from the C&K suite (Chidamber and Kemerer, 1994)) have been used to explore the relationship between class/system structure and test complexity. Some strong and significant relationships between several complexity- and size-related metrics of production code and internal test code properties have been found (e.g., (Bruntink and van Deursen, 2006, Zhou et al., 2012)).

However, as far as could be ascertained from the systematic mapping study, all previous research addressing class testability has used only static software measures. As explained in Chapter 4, it has been noted for some time that traditional static software metrics may be necessary but not sufficient for characterising, assessing and predicting the entire quality profile of OO systems (Basili et al., 1996). Additional characteristics of interest can be captured through the use of dynamic metrics. As described in Section 2.3.2., dynamic metrics have been shown to directly reflect the quality attributes of a system *in operation*. The work described in this chapter extends the investigation of software features as factors in class testability by characterising that code using dynamic metrics. A fuller discussion of dynamic metrics and their relative advantages over static metrics is presented in Chapters 2 and 4.

In this chapter, the relationships between dynamic software properties and class testability are investigated. In particular, two dynamic concepts are investigated: *Dynamic Coupling (i.e. highly coupled classes*) and *Key Classes (i.e. frequently executed classes)*. The two concepts are contended to be related to class testability, as described in the section that follows.

6.2 Testability Concepts

6.2.1 Dynamic Coupling

Dynamic, instead of static, coupling is used to measure coupling between classes, for the many advantages of this group of metrics over the static ones (See Section 2.3.2). As shown in Section 4.4.1, the measurement of dynamic coupling was found to be the most widely investigated topic in the literature on dynamic metrics. Although dynamic coupling has been used to measure several quality attributes, no previous work have attempted to relate dynamic coupling to testability. Section 2.3.2 provides a detailed overview on how dynamic coupling metrics can provide more insight compared to other traditional coupling metrics.

For the purposes of this work, dynamic coupling metrics that capture coupling at the object level are used. The specific measure of coupling used here is based on runtime method invocations/calls, and also on the direction of the invocation: two classes, class A and class B, are said to be coupled if a method from class A (*caller*) invokes a method from class B (*callee*), or vice versa. This relationship is described as a 'client-server' relationship: a 'client' class imports services from a 'server' class.

Therefore, coupling is measured in the following two forms (i.e., to account for both *callers* and *callees*):

1) When a class is accessed by another class at runtime, and
2) When a class accesses other classes at runtime.

To measure these levels of coupling we select the previously defined *Import Coupling (IC)* [also known as Efferent Coupling (EC)] and *Export Coupling (EC)* [also known as Afferent Coupling (AC)] metrics (Arisholm et al., 2004).

The ***IC*** for class A is the number of method invocations/calls ***received*** by class A (*callee*) from other classes (*callers*) in the system (4). This metric is also referred to as IC_CC (Import Coupling, Class-level, Distinct Class).

The ***EC*** for class A is the number of method invocations/calls ***sent*** from methods within class A to other classes (*callees*) in the system (5). This metric is also referred to as EC_CC (Export Coupling, Class-level, Distinct Class).

For both the IC and EC metrics, all invocations to and from methods within the same class (i.e., cohesive methods) are excluded. Also, multiple invocations to the same class are counted as a single (*one*) coupling. The formal definition of both metrics are given as follows (Arisholm et al., 2004):

$$IC_{C_1} = \{(m_1, c_1, c_2) | (\exists (m_1, c_1), (m_2, c_2) \in R_{MC}) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\} \quad (4)$$

$$EC_{C_1} = \{(m_2, c_2, c_1) | (\exists (m_1, c_1), (m_2, c_2) \in R_{MC}) \wedge c_1 \neq c_2 \wedge (m_2, c_2, m_1, c_1) \in IV\} \quad (5)$$

Where

C: set of classes in a systems.

M: set of methods in a system (as identified in each class in the system).

R$_{MC}$: set of all methods that are defined in a class: $R_{MC} \subseteq M \times C$.

IV: set of all possible method invocations in the system: $IV \subseteq M \times C \times M \times C$. An invocation is characterized by the invoking class and the class that has been invoked.

6.2.2 Key Classes

The notion of a *Key Class* is introduced in this study as a new production code property to be measured and its relationship to class testability investigated.

OO programs are formed around groups of classes that interact with each other. In typical software development, the number of classes increases as software systems and programs grow in size. To analyse and understand a software program, how it works, its potential for decay and its need for repair, it is important to know where to start and which aspects should be given priority. From a program comprehension and software maintenance perspective, understanding the roles of classes and their relative importance in a system is essential. In this respect, there are classes that could have more influence and play more prominent roles in the program design and architecture than others. In this thesis this group of classes is referred to as *'Key Classes'*. We define a *Key Class* as a class that is executed frequently in the typical use profile of a system. Identifying these classes should inform more effective planning. One of the potential uses of these classes is in prioritising testing activities – quality assurance personnel and software testers could usefully prioritise their work by focusing on testing these Key Classes first, alongside consideration of other factors such as risk and criticality information.

The concept of a key class has been used in a different way in previous work (e.g., (Tahvildar and Kontogiannis, 2004, Zaidman and Demeyer, 2008)). For example, in the work of Zaidman and Demeyer (2008), classification as a key class is based on the level of coupling of a class: key classes are those that are tightly coupled. In contrast, the definition presented in this work is based on the ***usage*** of these classes: Key Classes are those that have high execution frequency at runtime. The goal here is to examine whether Key Classes (i.e., those classes with higher frequency of execution) have a significant relationship with class testability. A new dynamic metric called *Execution Frequency (EF)* is proposed to identify and locate those Key Classes (6). *EF* for class *C* counts the number of executions of methods within class *C*, excluding self-calls.

Consider a class *C*, with methods m_1, m_2,..... m_n. Let EF(mi) be the number of executions of method *mi* of class *C*, then:

$$\mathrm{EF}(C) = \sum_{i=1}^{n} \mathrm{EF}(mi) \tag{6}$$

where $\boldsymbol{n}$ *is the number of executed methods within class* $\boldsymbol{C}$

6.3 Experimental Design

As explained in Chapter 3 (Section 3.3), metric selection in this research has been determined in a 'goal-oriented' manner. Our ***goal*** in this experiment is to better understand what affects class testability, and the ***objective*** is to assess the presence and strength of the relationships between dynamic complexity attributes (represented here by Dynamic Coupling and Key Classes) on the one hand and class testability (measured in terms of unit test size) on the other. The specific ***purpose*** is to measure and ultimately predict class testability in OO systems. The ***viewpoint*** is as software engineers, and more specifically, testers, maintainers and quality engineers. The targeted ***environment*** is Java OSS.

6.3.1 Research Questions and Hypotheses

The design above reflects the contention that two factors of interest are, in principle, related to class testability: Dynamic Coupling and Key Classes. To evaluate this contention the following research question is investigated:

$\mathbf{RQ_{6.1}}$: Is class complexity significantly correlated with class testability?

The following two research hypotheses are investigated to answer $RQ_{6.1}$:

$\mathbf{H_{6.1}}$: *Dynamic Coupling* has a significant correlation with unit test size.

$\mathbf{H_{6.2}}$: *Key Classes* have a significant correlation with unit test size.

The corresponding null hypotheses are:

NH$_{6.1}$: *Dynamic Coupling* has no significant correlation with unit test size.

NH$_{6.2}$: *Key Classes* have no significant correlation with unit test size

6.3.2 Data Collection and Execution Scenarios

As explained in Chapter 2, the collection of dynamic metrics data can be accomplished in various ways. The most common (and perhaps the *most* accurate) way is to collect the data by obtaining trace information using dynamic analysis techniques during software execution. Such an approach is taken in this study and is implemented by collecting metrics using the *AspectJ* framework, a well-established Java implementation of AOP. Previous works (including those of Cazzola and Marchetto (2008), Adams et al. (2009) and Tahir et al. (2010)) have shown that AOP is an efficient and practical approach for the objective collection of dynamic metrics data, as it can enable full runtime automatic source-code instrumentation to be performed. For coupling metrics, we used the same AOP (AspectJ) rules explained in Section 5.2.3 to collect these metrics. For EF metric, we use the following AspectJ rules:

```
pointcut capture_execution(): execution (* *..*(..)) // capture the executed classes

pointcut exclude(): ! execution (system-under-examination) // excludes all classes executions
that are not relevant to the program under examination such as the execution of default java
and AspectJ compiler's classes

executedClass_count  = 0; // for each executed class

before (): capture_execution() &&  exclude()
{
        // capture the name of the executed class
         executedClass = thisJoinPoint.getSignature().getDeclaringTypeName();
        // count the frequency of the execution (for each executed class)
        executedClass_count = count + 1;
}
```

In this study four OSS are selected for examination: FindBugs, JabRef, Dependency Finder and MOEA. Information about the selection process of these systems is shown in Section 3.2.1. General characteristics of the selected systems are provided in (Tables 3 and 5). As explained in Section 5.3, execution scenarios are designed to use key system features, based on the available documentation and user manuals for each system, as well as any prior knowledge of the systems held by those running the scenarios. Details of the execution scenario for each system are now explained.

FindBugs: the tool is run to detect bugs in a large-scale OSS (i.e., JFreeChart) by analysing the source code and the associated JAR files. The web plugin was installed during the execution and data were uploaded to the FindBugs webserver. Results were stored using all three file formats supported.

JabRef: the tool is used to generate and store a list of references from an original research report. All reference types supported by the tool were included (e.g., journal articles, conference proceedings, reports, standards). Reports were then extracted using all available formats (including XML, SQL and CSV). References were managed using all the provided features. All additional plugins provided at the tool's website were installed and used during this execution.

Dependency Finder: this scenario involved using the tool to analyse the source code of four medium- to large-sized systems one after another, namely, FindBugs, JMeter, Ant and Colossus. Dependencies were computed and depicted (in dependency graphs) and OO metrics at all layers (i.e., packages, classes, features) were calculated. Analysis reports on all four systems were extracted and saved individually.

MOEA: *MOEA* has a GUI diagnostic tool that provides access to a set of 6 algorithms, 57 test problems and search operators. This tool was used to apply the different algorithms to the predefined problems. Each of these algorithms was applied at least once on each problem. Metrics and performance indicators for all results provided by those different problems and algorithms were displayed. Statistical results of these multiple runs were displayed at the end of the analysis.

6.4 Results

On applying the Shapiro-Wilk test to the collected data for all four systems, the results confirmed that the data were not normally distributed (see Figures 19 and 20). Therefore, it was decided to use *Spearman's ρ* correlation test (*Spearman's ρ* is explained in more detail in Chapter 3 (Section 3.2.2)). In this work Spearman's ρ is calculated to assess the degree of association between each dynamic metric of the production code (i.e., IC, EC and EF) and the class testability metrics (defined in Section 3.3.2).

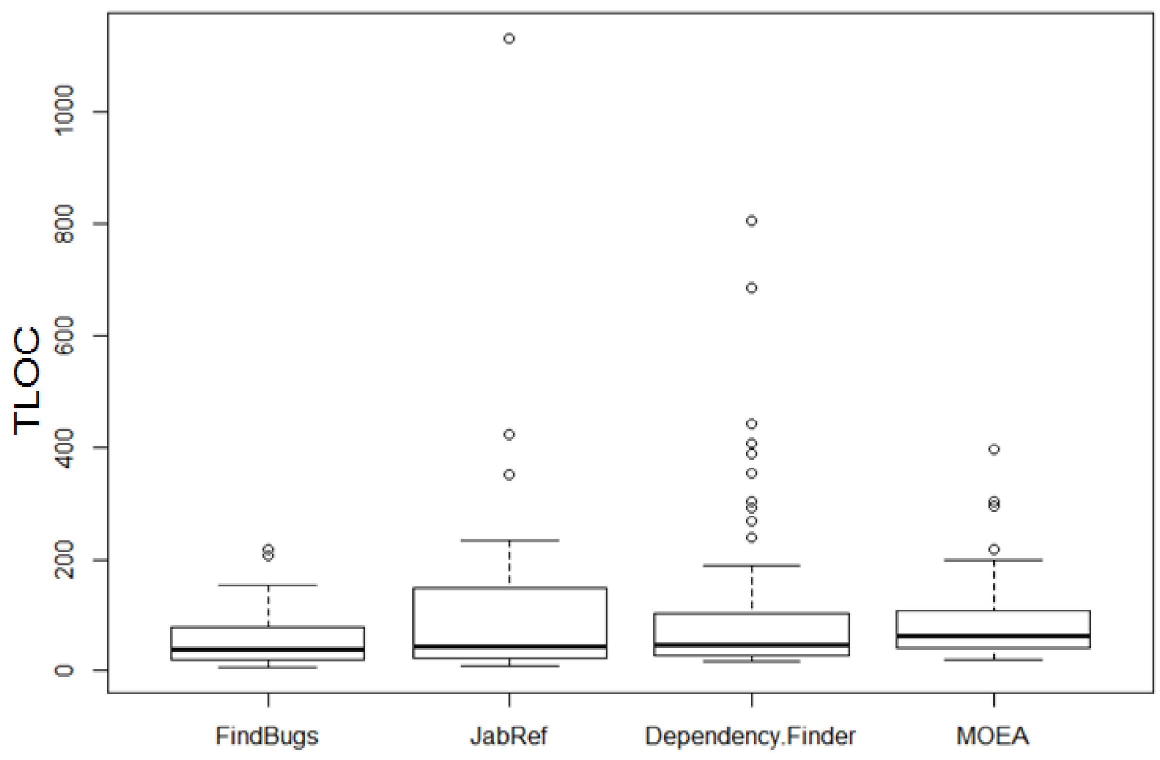


Figure 19. Boxplots of TLOC in all four system.

The number of observation points[37] considered in each test varies in accordance with the systems' execution scenarios described in Section 6.3.2. The numbers of observations are: *FindBugs* (23), *JabRef* (26), *Dependency Finder* (80) and *MOEA* (76). The total number of unit tests for each of these systems is shown in Table 4 (Chapter 3).

Table 20 shows the *Spearman's ρ* results for the two dynamic coupling metrics against the class testability metrics. Corresponding results for the EF metric against the test suite metrics are presented in Table 21. For all analyses, it is interpreted that there is a significant correlation between two variables if there is statistically significant evidence of such a relationship in at least ***three*** of the ***four*** systems examined. All significant values in Tables 20, 21 and 22 are marked with an asterisk (*) and all *medium* and *high* correlations are shown in **Bold**

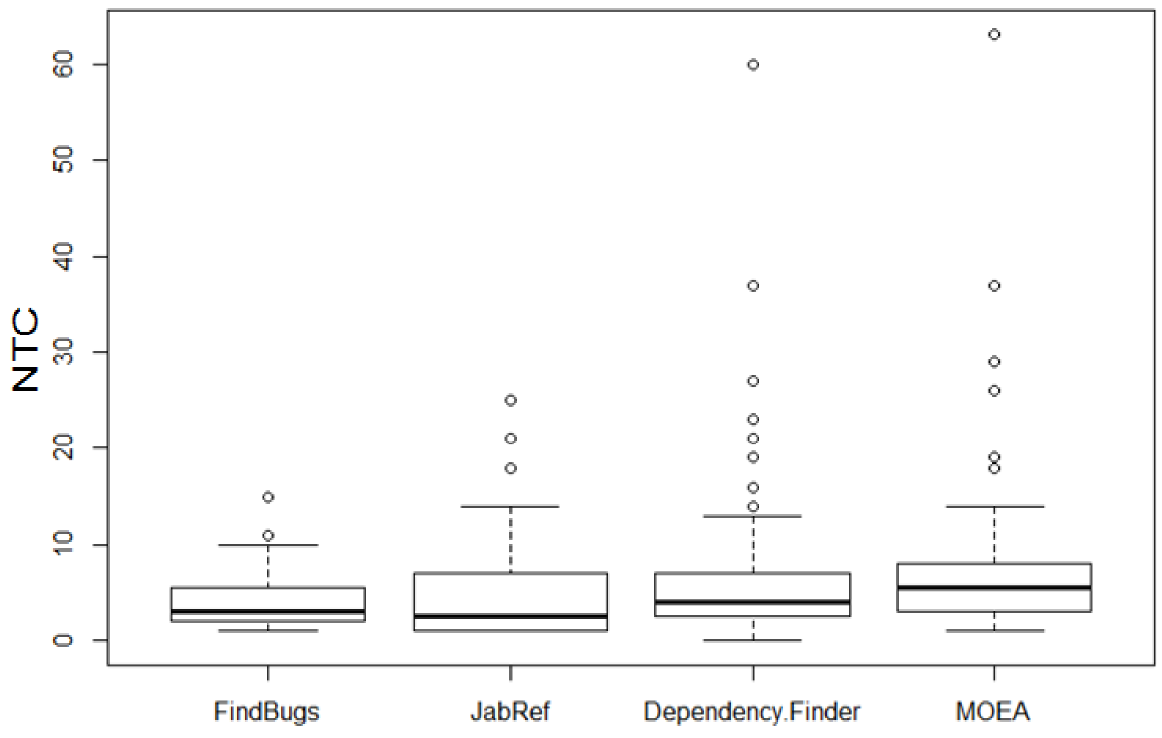


Figure 20. Boxplots of NTC in all four systems

[37] The number of observation points here is represented by the number of the tested classes that were traversed in the execution (i.e., classes that have corresponding tests and that were captured during the execution by any of the dynamic metrics used).

As is evident in Table 20, EC is observed to have a significant medium to high correlation with the TLOC metric in all four systems. The correlation was found to be high in *Dependency Finder* and medium in *FindBugs*, *JabRef* and *MOEA*. A similar significant correlation between EC and NTC is evident in three of the four systems: *FindBugs* (*high* association), *JabRef* and *Dependency Finder* systems (both *medium* associations). In terms of relationships with the IC metric, a significant correlation between IC and TLOC is evident only in one system (*high* association in *Dependency Finder*). For the relationship between IC and NTC, a direct medium correlation was found only in one system i.e., *Dependency Finder*. A low inverse association between IC and NTC is evident for the *MOEA* system.

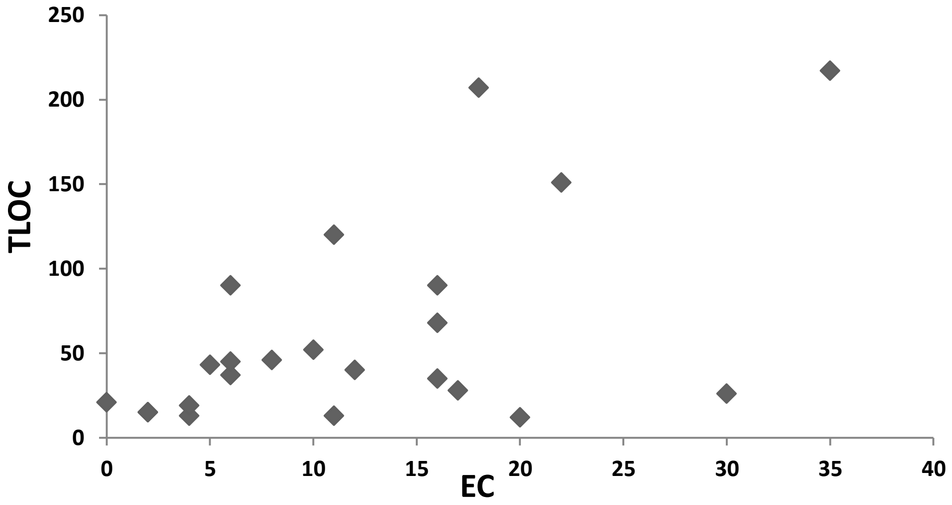
Table 20. Spearman's *ρ* correlations between dynamic coupling and class testability metrics

Systems	Metrics	TLOC		NTC	
		ρ	p-value	*ρ*	p-value
FindBugs	EC	**.43**	**.04***	**.58**	**.00***
	IC	-.07	.77	-.09	.69
JabRef	EC	**.35**	**.04***	**.33**	**.05***
	IC	.28	.09	.23	.13
Dependency Finder	EC	**.52**	**.00***	**.41**	**.00***
	IC	**.52**	**.00***	**.33**	**.00***
MOEA	EC	**.30**	**.01***	.12	.16
	IC	-.08	.24	-.24	.02*

As shown in Table 21, positive significant associations were found between EF and the class testability metrics in three of the four systems (the exception being *MOEA*). A significant *medium* correlation between EF and TLOC was found in *FindBugs*, *JabRef* and *Dependency Finder*. Also, a *medium* correlation between EF and NTC was found in *JabRef*, and a *low* correlation is found in *Dependency Finder*.

Table 21. Spearman's *ρ* correlations between EF and class testability metrics

Systems	Metrics	TLOC		NTC	
		ρ	p-value	*ρ*	p-value
FindBugs	EF	**.42**	**.05***	.37	.09
JabRef	EF	**.44**	**.01***	**.38**	**.03***
Dependency Finder	EF	**.33**	**.00***	.22	.03*
MOEA	EF	.03	.41	-.10	.19



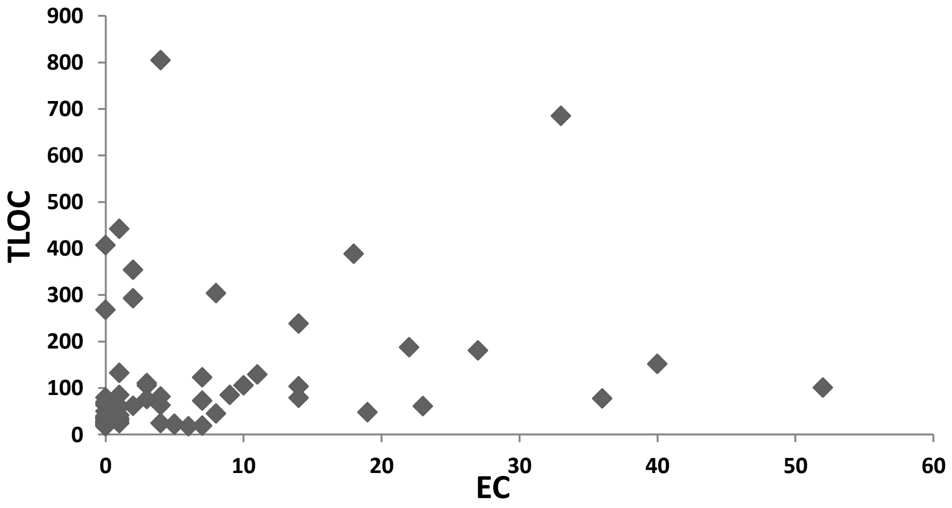
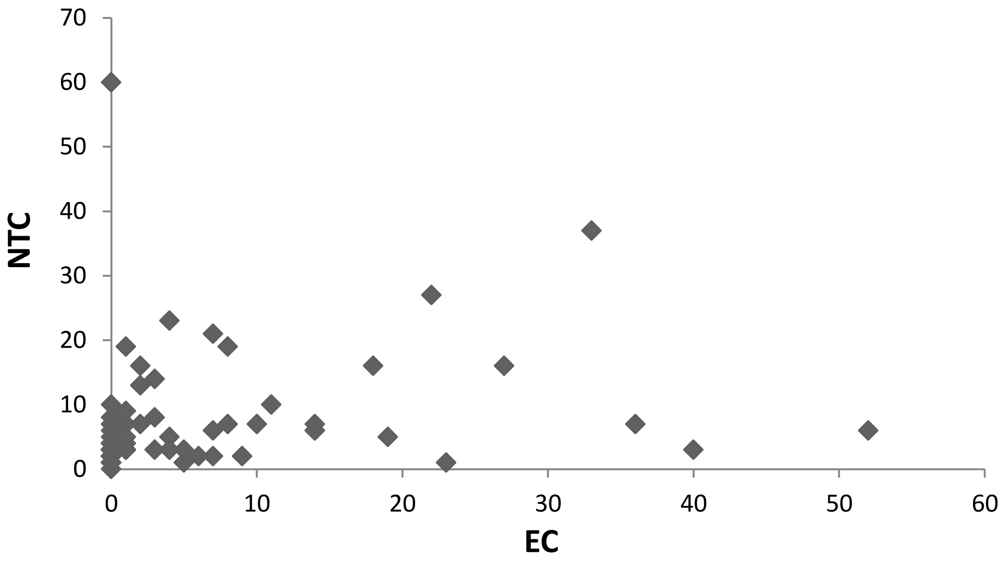


Figure 21. Scatter plot of the relationship between EC and TLOC in *Dependency Finder* (top) and *FindBugs* (bottom)



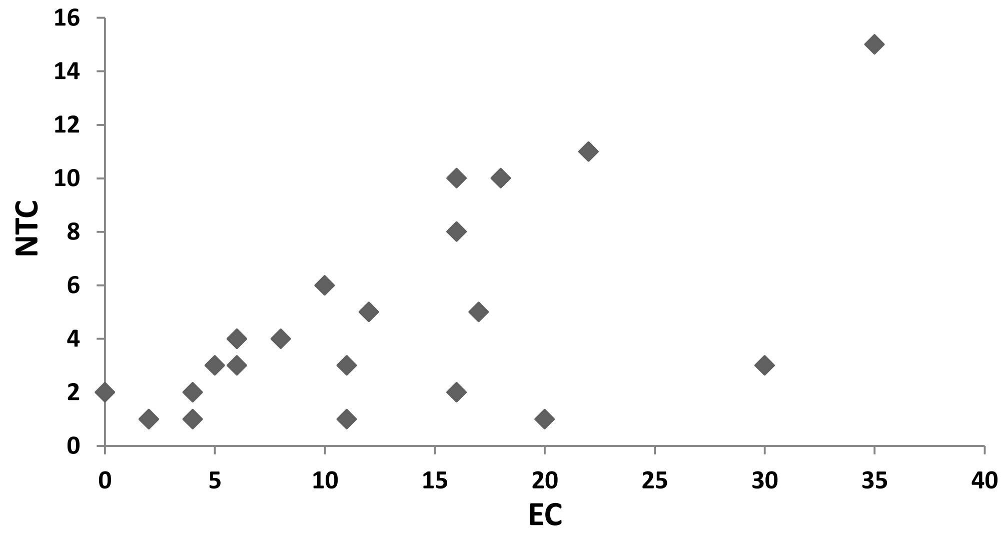


Figure 22. Scatter plot of the relationship between EC and NTC in *Dependency Finder* (top) *and FindBugs* (bottom)

Based on our analysis $H_{6.1}$ is accepted and $NH_{6.1}$ is rejected; that is, there is evidence of a significant association between dynamic coupling (either EC or IC) and the two class-testability metrics for all four systems. As EF is also found to be significantly associated with the testability metrics for three of the four systems considered, $H_{6.2}$ is also accepted and $NH_{6.2}$ is rejected on the balance of evidence. *Scatter Plot* graphs of the most significant (and highly associated) correlations are now depicted. Figure 21 shows *Scatter Plots* of the relationship between EC and TLOC in *Dependency Finder* and *FindBugs*. Figure 22 shows *Scatter Plots* of the relationship between EC and NTC in *FindBugs* and *Dependency Finder.* Scatter plots of the relationship between EF and TLOC in *JabRef* and *FindBugs* are shown in Figure 23. Note that these graphs are from the systems with the most significant (and highly associated) correlations (this also applies to Figures 21-24). Graphs for all other significant correlations are provided in Appendix D. As shown in these graphs, there are a number of outlier and leverage points that should be taken into consideration if a prediction model is developed based on these results.

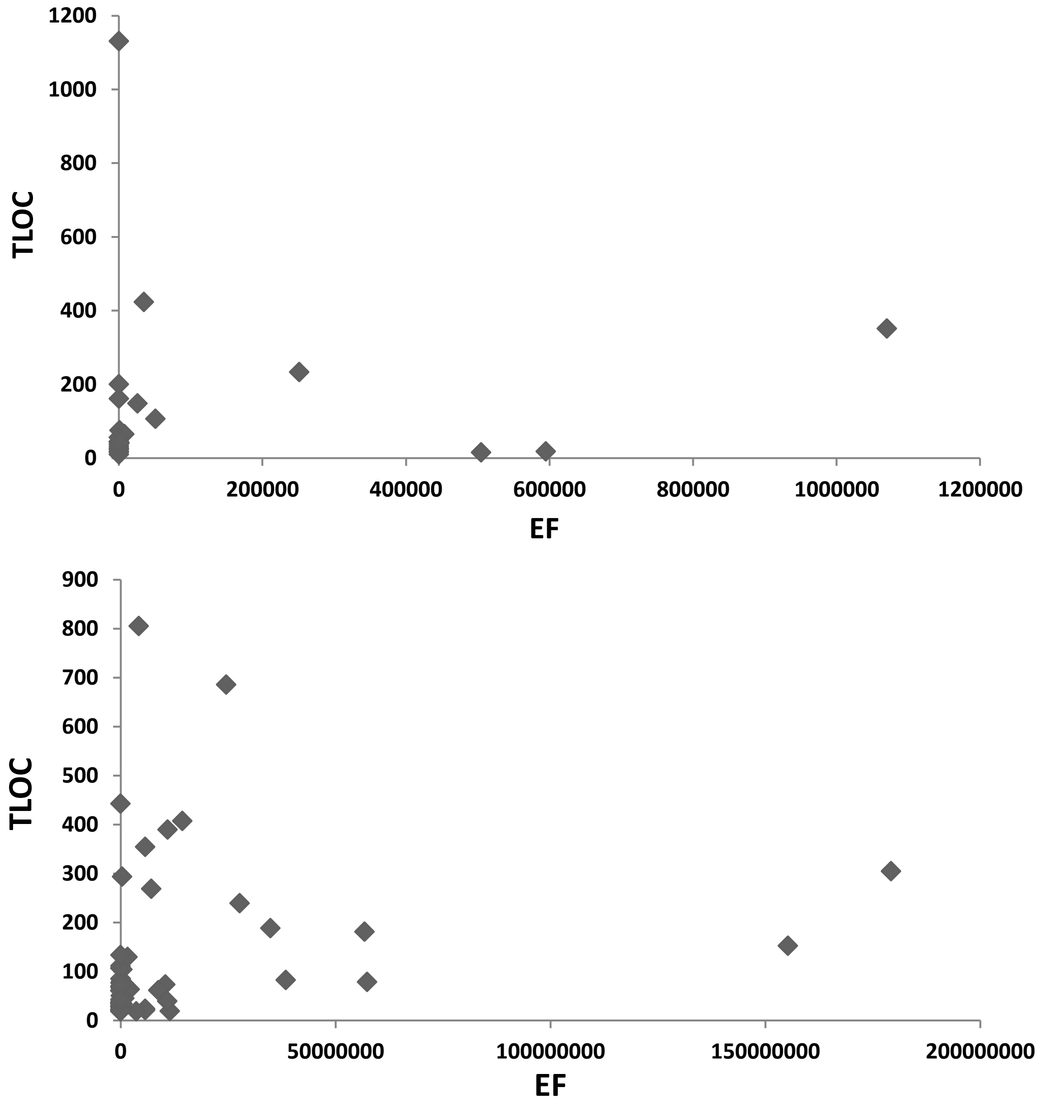
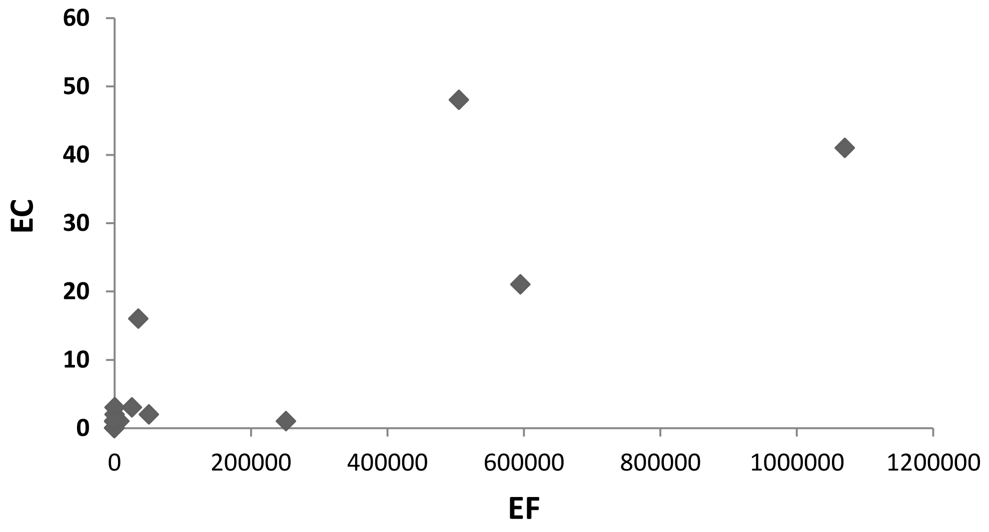


Figure 23. Scatter plot of the relationship between EF and TLOC in *JabRef* (top) and *Dependency Finder* (bottom)

Another test of relevance in this experiment is to consider whether the dynamic metrics used are themselves correlated, since this may indicate that only a subset of these metrics needs to be collected. Therefore, a further correlation analysis was performed to investigate this correlation. The results show that the two dynamic coupling metrics used here (i.e., IC and EC) are correlated with EF to varying degrees for the four systems investigated (with Scatter Plots shown in Figure 24). High direct and medium direct associations between EC and the EF metric are evident in three systems (the only exception is *FindBugs*). IC is correlated with EF in only two systems (high correlation in *Dependency Finder* and low in *MOEA*).



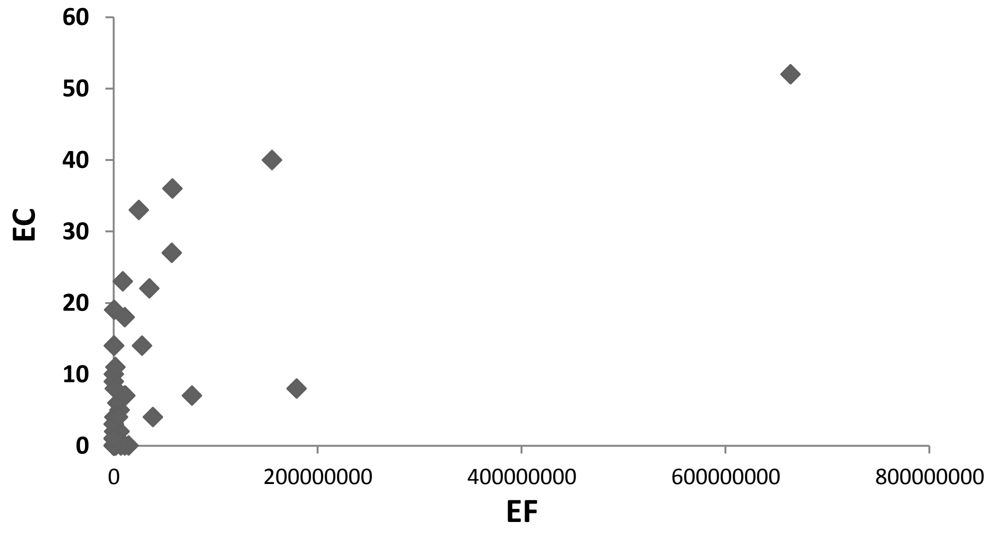


Figure 24. Scatter plot of the relationship between EF and EC in *JabRef* (top) and *Dependency Finder* (bottom)

6.5 Discussion

The outcomes of this experiment indicate that complexity attributes that can be measured through dynamic analysis are indeed associated with class testability.

There is statistical evidence of a significant association between dynamic coupling (via either the EC or IC metric) and unit test size in all four OSS that were examined in the experiment. Coupled classes are 'connected' with other classes within the program. It seems plausible to assert that, when writing a unit test to test a tightly coupled class, a developer/tester will require adequate coverage of the code. This might mean a need to include all other dependent components related via method calls/invocations made from and to these classes. This would result in an increase in the size of the unit test, due to the extended connections that the unit test might need to cover. Additionally, *Key Classes* (i.e., frequently executed classes) is also found to be significantly associated with the testability metrics collected, for three of the four systems considered. This suggests that tightly coupled classes *and* classes that are executed frequently in genuine use scenarios for a program are more likely to have larger unit tests compared to other classes within a system.

Thus, in revisiting the list of the investigated research questions, dynamic coupling has been found to have a significant (although not very strong) direct association with class testability ($H_{6.1}$). A more significant correlation was found between key classes (i.e., frequently executed classes) and class testability metrics ($H_{6.2}$). By answering $RQ_{6.1}$, this suggests that dynamic coupling and key classes can act, to some extent, as initial complementary indicators of class testability. It is contended here that a tightly coupled or frequently executed class would need a large corresponding unit test (i.e., higher numbers of TLOC and NTC). Such results could be helpful for testers and maintainers as they provide empirical evidence regarding the relationship between two important dynamic properties and class testability. We recommend that similar dynamic information should be taken into consideration when developing unit tests or maintaining existing unit tests. While further testing is needed we would at this stage conclude that the two dynamic properties examined should be used ahead of static properties as they provide a more comprehensive indication of class testability.

An additional test of relevance in this study is to consider whether the dynamic metrics used are themselves related, since this may indicate that only a subset of these metrics should be collected. Therefore, a further correlation analysis was performed to investigate this. The results indicate that the two dynamic coupling metrics are correlated with EF (Table 22) to varying degrees for the four systems investigated. That is, Key Classes are also associated with dynamically coupled classes: a Key Class is most likely to be a tightly coupled class, and vice-versa. Which particular metrics should be used in a given analysis is an appropriate subject of future study.

Table 22. Spearman's *ρ* results for the correlation between coupling and EF dynamic metrics

Systems	Metrics	IC		EC	
		ρ	p-value	ρ	p-value
FindBugs	EF	.32	.14	.31	.15
JabRef	EF	.27	.09	**.81**	**.00***
Dependency Finder	EF	**.56**	**.00***	**.51**	**.00***
MOEA	EF	.29	.01*	**.40**	**.00***

6.6 Threats to Validity

A number of threats that could affect the validity of the results of this experiment are acknowledged in the following. Note that the nature of these threats is explained in more detail in Section 3.5.1.

Ambiguity about the Direction of Causal Influence

In this chapter, the directions and impacts of the correlation between individual variables were not investigated (directly). This thesis makes assumptions of directionality (impact or cause-effect) in the correlations between different variables based on the theory and findings identified in previous research. However, it has not been considered if there are other factors that can cause the discovered correlations. Obtaining testing strategy information can be helpful to reduce this threat. Such information can help to identify which pieces of the code were developed first.

Generalisation of Findings

One of the possible threats to the validity of this study is the limited number of systems used in the analysis. The results discussed here are derived from the analysis of four open source systems. The consideration of a larger number of systems, perhaps also including closed-source or industrial based systems, could enable further evaluation of the associations revealed in this study and so lead to more generalizable conclusions.

Selection of the Execution Scenarios

One of the possible threats is the subjective selection of the execution scenarios. This threat and its mitigation have been discussed in more detail in Section 3.5.1.

6.7 Summary

In this chapter the presence and significance of associations between two runtime code properties, namely Dynamic Coupling and Key Classes, and the internal testability of classes has been investigated using four OSS. Class testability was measured using two size metrics, namely TLOC and NTC, while Dynamic Coupling and Key Classes were measured using dynamic software metrics collected via AOP. Correlations were analysed statistically using the Spearman's ρ test to study the strength of the associations.

The resulting evidence indicates that there is a significant association between Dynamic Coupling and internal class testability – Dynamic Coupling metrics, and especially EC, have a significant direct association with TLOC. A less significant association was found between IC and NTC. Similarly, Key Classes are also shown to be significantly associated with the class testability metrics in two of the three systems examined.

The following Chapter looks in more detail at the issue of code and test smells and their impact on both unit test and production classes.

Chapter 7 On the Quality of Unit Tests

The Impact of Test Smells

7.1 Introduction

Society's growing reliance on software has led to increased research attention being directed to the study, and prevention, of software quality issues as indicated by smells. Researchers have been working for several years to provide empirical evidence of the impact of *code* smells on software artefacts and processes. In contrast, there has been relatively little attention given to the study of *test* smells and their impact on software artefacts and activities. Test smells, as with code smells, can arise as a result of poor design or implementation of a unit test. Researchers have used the term *test smells* to refer specifically to the group of code smells that affect only unit tests. The term was first defined by van Deursen et al. (2001) and was further explained by Meszaros (2006).

The primary motivation of this particular experiment was to address the so far limited coverage given to test smells by providing an in-depth empirical investigation into the factors that may impact the presence of smells in a unit test, from a range of perspectives. Specifically, the experiment presented in this chapter investigates the relationship between test smell types (individually and collectively) and:

1) the size of the unit test,
2) the size and complexity of the associated production class,
3) the co-occurrence of test and code smell types, and
4) the co-occurrence of test smells (i.e., how often the presence of a test smell type in a unit test implies the presence of another test smell type).

through nine research questions. This experiment is motivated by recent work in test and code smells which has suggested the need for more comprehensive studies to investigate the impact of smells on software systems (Zhang et al., 2011, Bavota et al., 2014).

The experimental design is presented in the following section.

7.2 Experimental Design

The ***goal*** of this experiment is to investigate the relationships between test smells and the size, complexity and presence of code smells in classes. Note that these relationships are assessed at class-level (i.e., relationships between a unit test and its associated production class). The ***context*** of the work is unit tests and their associated production classes in OSS. The ***quality focus*** is to reduce the effort required in understanding and maintaining unit tests during software maintenance. The ***viewpoint*** of this work is from both researchers and practitioners who are seeking to understand the design and source code factors that may result in test smells in a unit test.

7.2.1 Research Questions and Hypotheses

The experiment presented in this chapter investigates a number of research questions that are related to test smells and/or class testability. The main aim of this chapter is to improve overall understanding of the relationships between test smells and software characteristics on the one hand, and between test smells and code smells on the other. In this experiment the relationships between 9 different test smells and 10 other code smells are therefore explored. Tables 23 and 24 provide full descriptions of the test and code smells respectively that are considered in this experiment. These smells were selected because they are well-defined in the literature, they have been considered in similar previous studies (so their descriptions are clear), and there were tools available (a mix of academic and commercial tools) to detect these smells. The possible impact of these test and code smells, and suggested refactoring techniques to eliminate their impact, are explained in detail in van Deursen et al. (2001) and Fowler et al. (1999) respectively.

Table 23. List of test smells and the detection tools used

Test Smell	Description	Tool
Assertion-free	A test case without assertion	PMD
Assertion Roulette	Test method having more than one assertion	PMD
Sensitive Equality	The *toString* method is used in assert statements	(Bavota et al., 2014)
Mystery Guest	A test case that uses external resources.	(Bavota et al., 2014)
Indirect Test	A test allocates resources also used by others	(Bavota et al., 2014)
General Fixture	A test fixture is too general and the test methods access only part of it	(Bavota et al., 2014)
Duplicated Code	Sets of test commands that contain the same invocation and data access sequence.	CodePro
Eager Test	A unit test has at least one method that uses more than one method of the tested class	(Bavota et al., 2014)
Lazy Test	Several test methods check a method of the tested class using the same fixture	(Bavota et al., 2014)

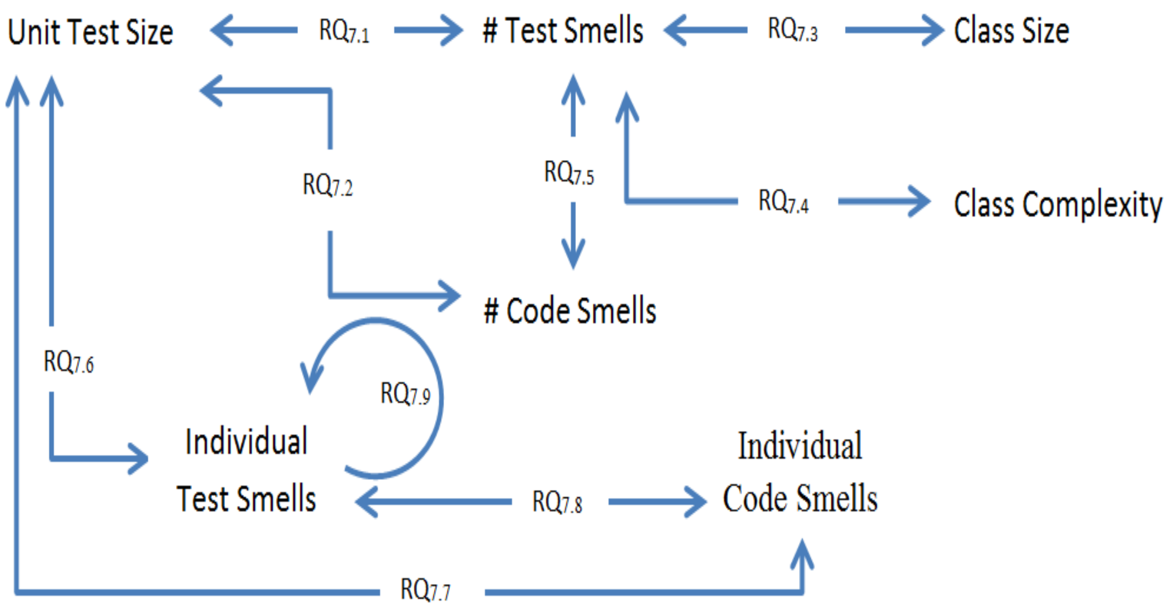


Figure 25. Overview of research questions and the link between them

Table 24. List of code smells and the detection tools used

Code Smell	Description	Tool
Feature Envy	A method making too many invocations to methods of another class	JDeodorant
Large Class	A class with at least one large method (in terms of LOC) compared to the other methods within the class	CodePro
Type Checking	A complicated conditional statement within a class	JDeodorant
Brain Class	Too complex class that takes too much responsibilities	JDeodorant
Duplicated Code	Fragments of source code that appear in more than one place in the program	CodePro
Schizophrenic Class	A class with a large and non-cohesive interface	InCode
Data Class	A class with an interface that exposes data members, instead of providing any substantial functionality	InCode
Data Clumps	Large groups of parameters that appear together in the signature of many operations	InCode
Tradition Breaker	A class that breaks the interface inherited from a base class or an interface	InCode
Message Chain	An operation that accesses a sequence of data exposer members from other classes to hop between multiple objects	InCode

Table 25 lists all of the research questions and hypotheses investigated in this chapter, and Figure 25 provides a high-level overview of the relationships between the research questions. The first point of investigation is whether the size of a unit test has an impact on the number of smell types (both test smells [$RQ_{7.1}$] and code smells [$RQ_{7.2}$]). The first question specifically considers whether the number of test smell types increases with the size of the unit test ($RQ_{7.1}$). The recent work of Bavota et al. (2014) studied the relationships between the number of test smells in a system and several system size measures including LOC and NOC, number of unit tests and the size of the development team in a project. In contrast, the focus in this thesis is on relationships at the class-level rather than the system level (i.e., relationships between smells in individual unit tests and their associated production classes). This should enable greater precision, as it enables us to investigate the relationship between artefact size and test smells at a smaller, more fine-grained scale (via the relationship between unit tests and production classes). As explained in Section 3.3.2, the size of a unit test was measured here using the two testability metrics TLOC and NTC. For $RQ_{7.1}$, the following hypothesis $H_{7.1}$ is tested**:** *there is a significant positive relationship between the size of a unit test and the number of test smell types in that unit test.*

We then explore the relationship between the number of code smell types in a production class and the size of the corresponding unit test. Recently Sabane et al. (2013) found that classes that contain code smells require a higher number of test cases compared to classes without smells. This experiment addresses a similar research question: is the number of code smell types in the production class related to the size of the corresponding unit test (measured using NTC and TLOC) ($RQ_{7.2}$). However, while Sabane et al. (2013) used the MaDUM technique, which estimates the number of test cases required to test each production class, this work uses the *actual* number of test cases in the unit test. It is contended that using the actual number of test cases is likely to better reflect the true development practice in OSS. We test the following hypothesis for $RQ_{7.2}$- $H_{7.2}$**:** *there is a significant positive relationship between the number of code smell types present in a production class and the size of the unit test.*

Another aim of this work is to investigate whether static properties of a production class (i.e., size and complexity) have an impact on the number of test smells present in the corresponding unit test. This research question is motivated by the previous findings which explain how software size is related to many other software metrics (i.e., it is a confounding factor) (El Emam et al., 2001, Zhou et al., 2014). The following research question is investigated: is the number of test smell types in a unit test related to the size of the corresponding production class? (RQ$_{7.3}$). We investigate the following hypothesis for this research question $H_{7.3}$: *there is a significant positive relationship between the number of test smell types present in a unit test and the size of the corresponding production class*.

Table 25. Summary of the research questions and hypotheses

Research Question		Hypothesis	
RQ$_{7.1}$	Does the number of test smell types increase with the size of the unit test?	**H$_{7.1}$**	There is a significant positive relationship between the size of a unit test and the number of test smell types in that unit test.
RQ$_{7.2}$	Is the number of code smell types in the production class related to the size of the corresponding unit test?	**H$_{7.2}$**	There is a significant positive relationship between the size of the unit test and the number of code smell types in the associated production class.
RQ$_{7.3}$	Is the number of test smell types in a unit test related to the size of the corresponding production class?	**H$_{7.3}$**	There is a significant positive relationship between the number of test smell types present in a unit test and the size of the corresponding production class.
RQ$_{7.4}$	Is the number of test smell types in a unit test related to the complexity of the corresponding production class?	**H$_{7.4}$**	There is a significant positive relationship between the number of test smell types present in a unit test and the complexity of the corresponding production class.
RQ$_{7.5}$	Does the number of test smell types in the unit test increase with the number of code smell types in the corresponding production class?	**H$_{7.5}$**	There is a significant positive relationship between the number of test smell types in a unit test and the number of code smell types in the corresponding production class.
RQ$_{7.6}$	Does the size of the unit test vary among unit tests exhibiting different kinds of test smell types?	**H$_{7.6}$**	There is a significant difference in the size of unit tests that contain different kinds of test smell types.
RQ$_{7.7}$	Does the size of the unit test vary among classes exhibiting different kinds of code smell types?	**H$_{7.7}$**	There is a significant difference in the size of unit tests that contain different kinds of code smells in the associated production class.
RQ$_{7.8}$	Do individual test smell types in a unit test co-occur with particular code smells in the corresponding production class?	**H$_{7.8}$**	There is a significant relationship between the co-occurrence of individual test smell types and individual code smell types
RQ$_{7.9}$	Do individual test smell types co-occur with each other in the same unit test?	**H$_{7.9}$**	There is a significant relationship between the co-occurrence of individual test smell types.

As explained previously in Section 2.4.1, prior research (such as Bruntink and van Deursen (2006)) has shown how the complexity of a class can impact its testability.

We also explained in Chapter 6 how dynamic complexity measures of a class are correlated with its testability. Given that such relationships exist between class complexity and class testability, it is equally important to see if such relationships impact the presence of test smells in the unit tests. To address this, we investigate the following research question: is the number of test smell types in a unit test related to the complexity of the corresponding production class? ($RQ_{7.4}$). Class complexity is measured here based on the average cyclomatic complexity per class. For $RQ_{7.4}$**,** the following hypotheses are tested – $H_{7.4}$: *there is a significant positive relationship between the number of test smell types present in a unit test and the complexity of the corresponding production class*.

This experiment next considers whether the total number of test smell types in a unit test is correlated with the total number of code smell types in its corresponding production class. In particular, the work investigates the following question: does the number of test smell types in the unit test increase with the number of code smell types in the corresponding production class? ($RQ_{7.5}$). For this research question, the following hypothesis is tested $H_{7.5}$: *there is a significant positive relationship between the number of test smell types in a unit test and the total number of code smell types in the corresponding production class*.

The study then looks deeper into the relationship between smells (both test and code) and unit test size by inspecting this relationship in regard to all individual smells. The conjecture is that some smells are more associated with the size of the unit test than others, given the nature of some of the smells as they appear to be influenced by size. In particular, we investigate how the size varies among unit tests that contain different kinds of test smells, by addressing the following question: does the size of the unit test vary among unit tests exhibiting different kinds of test smell types? ($RQ_{7.6}$). We then look at the equivalent relationship between code smells and the size of the unit test by addressing the following question: Does the size of the unit test vary among classes exhibiting different kinds of code smell types? *(*RQ$_{7.7}$*)*. For RQ$_{7.6}$ and RQ$_{7.7}$, we investigate the following hypotheses respectively- *H$_{7.6}$: there is a significant difference in the size of unit tests that contain different kinds of test smell types,* and *H$_{7.7}$: there is a significant difference in the size of unit tests that contain different kinds of code smell types in the associated production class.*

The final aspect of this experiment examines the co-occurrence (or coexistence) of test and code smell types. The intent here is to check if any of the test smell types co-occur with particular code smell types. The experiment considers all individual test smell types and their correlation with individual code smell types: do individual test smell types in a unit test co-occur with particular code smell types in the corresponding production class? (RQ$_{7.8}$), expressed in the following hypothesis, *H$_{7.8}$: there is a significant relationship between the co-occurrence of individual test smell types and individual code smell types*.

Similarly, this experiment investigates the co-occurrence (coexistence) of test smell types among themselves. The following research question is examined: do individual test smell types co-occur with each other in the same unit test? (RQ$_{7.9}$) which is expressed in the following hypothesis, *H$_{7.9}$*: *there is a significant relationship between the co-occurrence of individual test smell types*. For both questions, we investigate a relationship between two binary variables.

7.2.2 Data Collection

In total, eight different OSS are selected for this experiment: JFreeChart, FindBugs, JMeter, JabRef, Apache Commons Lang, Dependency Finder, MOEA and Barcode4J. The selection process of these systems is explained in Section 3.2.1 and their general characteristics are shown in Tables 3, 4 and 5. These systems form around 635 KLOC and contain in total 4854 production classes and 1280 unit tests. Across the eight systems examined a total of 1100 unit tests are analysed in this experiment. (Note that we excluded all non-valid unit tests (180 unit tests in total) e.g., a unit test that does not *directly* test one single production class.) Boxplots of the numbers of test and code smell types respectively in all eight systems are shown in Figures 26 and 27.

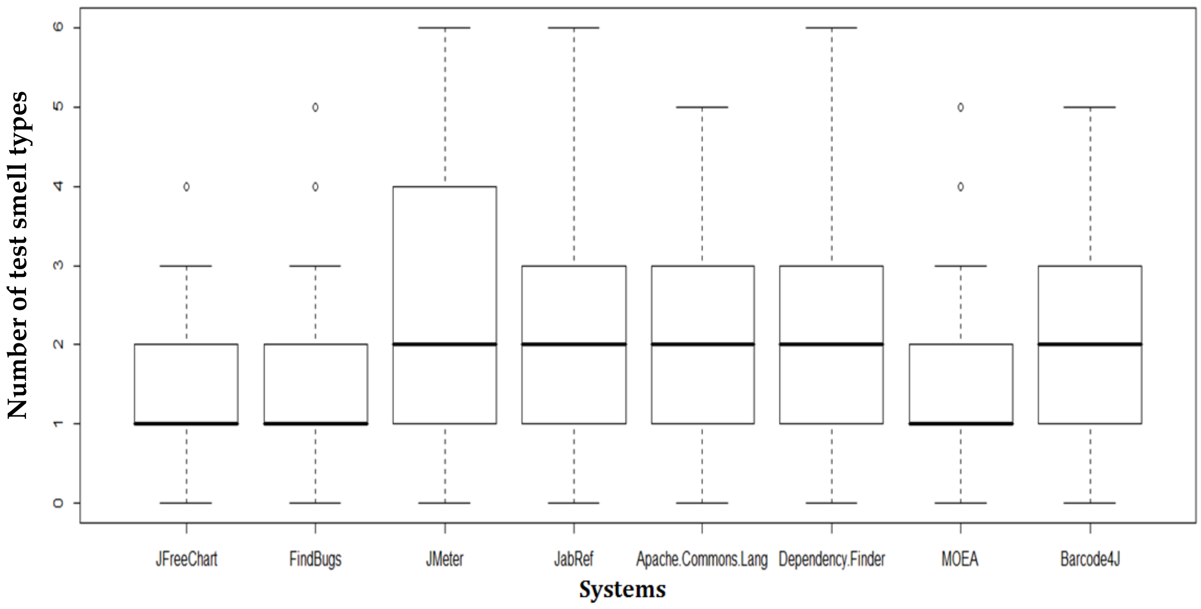


Figure 26. Boxplots of the number of test smell types per unit test in all systems

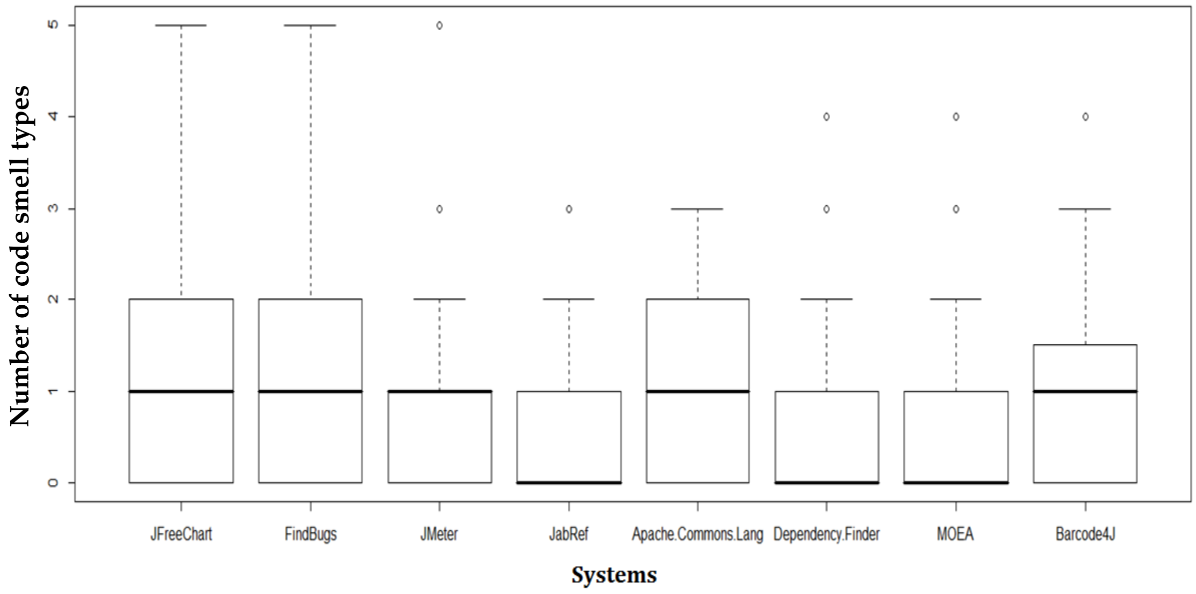


Figure 27. Boxplots of the number of code smell types per class in all systems

$H_{7.1}$- *$H_{7.5}$* are tested using the non-parametric *Spearman's rho (ρ) rank correlation coefficient* test (as per the detailed description of all the statistical tests used in this thesis provided in Section 3.2.2). *$H_{7.6}$* and *$H_{7.7}$* are examined using the *Mann-Whitney U test*. We also measure the *effect size (ez)* for this non-parametric data (see

Section 3.2.2). For $RQ_{7.8}$ and $RQ_{7.9}$, the relationships between two binary variables are investigated, therefore $H_{7.8}$ and $H_{7.9}$ are tested using the *Phi (ϕ) Correlation Coefficient* test.

All smells (both test and code) are captured using the tools shown in Tables 23 and 22. Only the *Feature Envy* code smell was detected using two tools. For this smell, the results from both JDeodorant and InCode were aggregated. We flagged a class as having *Feature Envy* if it is detected by either tool. We initially did this because we were not sure of the rules JDeodorant uses to identify *Feature Envy*. However, we found that over 90% of the reported *Feature Envy* instances were detected by both tools.

To improve the accuracy of the smells detected, a verification process was carried out through a manual inspection process. This was followed by cross-validation of some of the results with those obtained for the same systems considered in the prior work of Bavota et al. (2014) using the authors' publically available data[38] (for the 6 smells detected by the tool as shown in Table 23, i.e., *Sensitive Equality*, *Mystery Guest*, *Indirect Test*, *General Fixture*, *Eager Test* and *Lazy Test*). The cross-validation was conducted *manually* on the following systems[39]: FindBugs, JabRef, Dependency Finder and Barcode4J. In total, 224 unit tests were cross-validated, which corresponds to approximately 21% of the total number of the studied unit tests. Table 26 shows details of the total number smells in the cross-validated unit tests. These unit tests contain 143 instances of smells in total (and note that this only includes test smells detected by the tool). The cross-validation highlighted

[38] The results were obtained from the same tool. However, the authors conducted a manual inspection on all detected smells before performing any analysis on the obtained results.

[39] Note that the authors did not analyze all unit tests within the examined systems. For example, this work examined 220 unit tests for Dependency Finder, whereas Bavota et al. (2014) provided results for only 120 unit tests. The reason for reporting smells from only a selected number of unit tests was not explained in the original work of Bavota et al. (2014).

approximately 12% false positive smells. All false positive results were rechecked and revalidated manually, before they were removed

Table 26. Cross-validation results

System	#Analysed unit tests	#Cross-validated unit tests	Total number of false positive
FindBugs	38	26	4
JabRef	56	49	6
Dependency Finder	220	118	10
Barcode4J	31	31	6

The relative proportions of each individual test and code smell in all eight systems are shown in Figures 28 and 29 respectively, and the corresponding numbers of test and code smells are presented in Tables 27 and 28. Figure 30 shows the proportion of unit tests and production classes that contain at least one smell across all eight systems. The following section presents the results of the experiment conducted in this chapter.

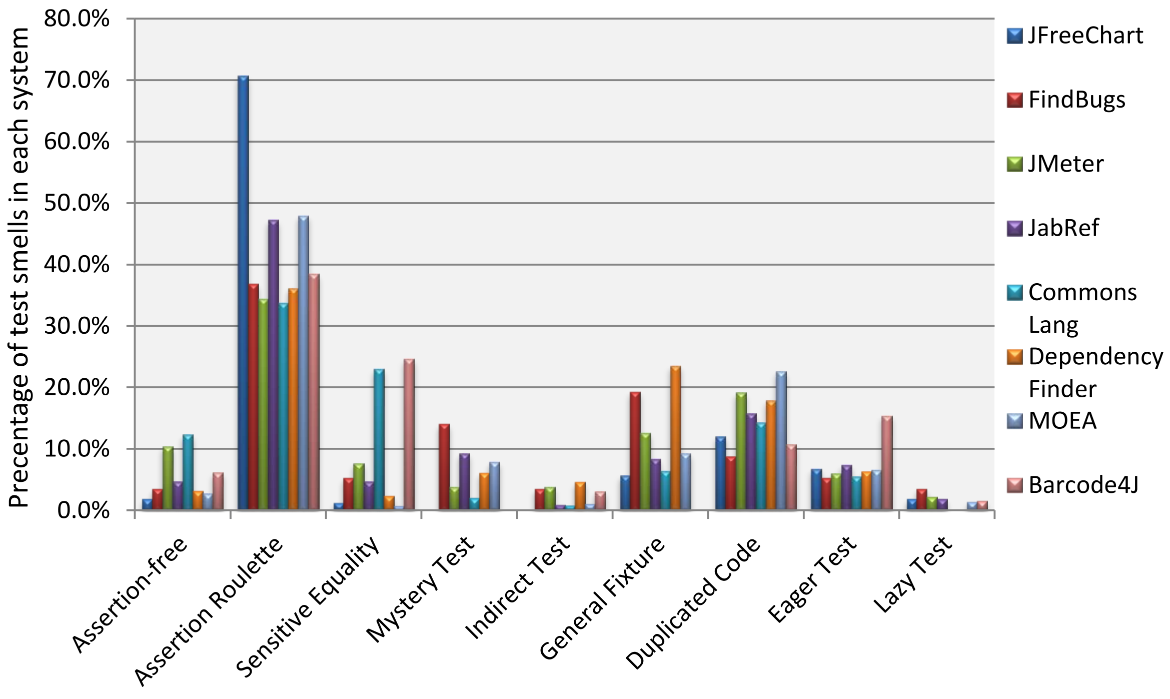


Figure 28. Distribution of test smell types in all eight examined systems

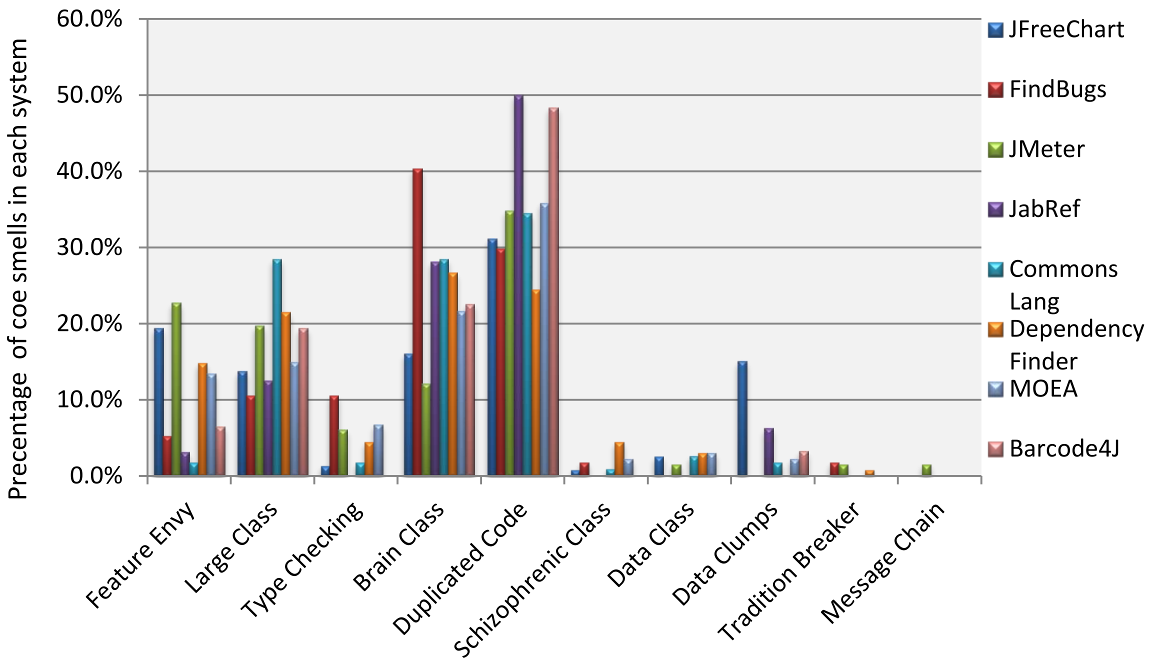


Figure 29. Distribution of code smell types in all eight examined systems

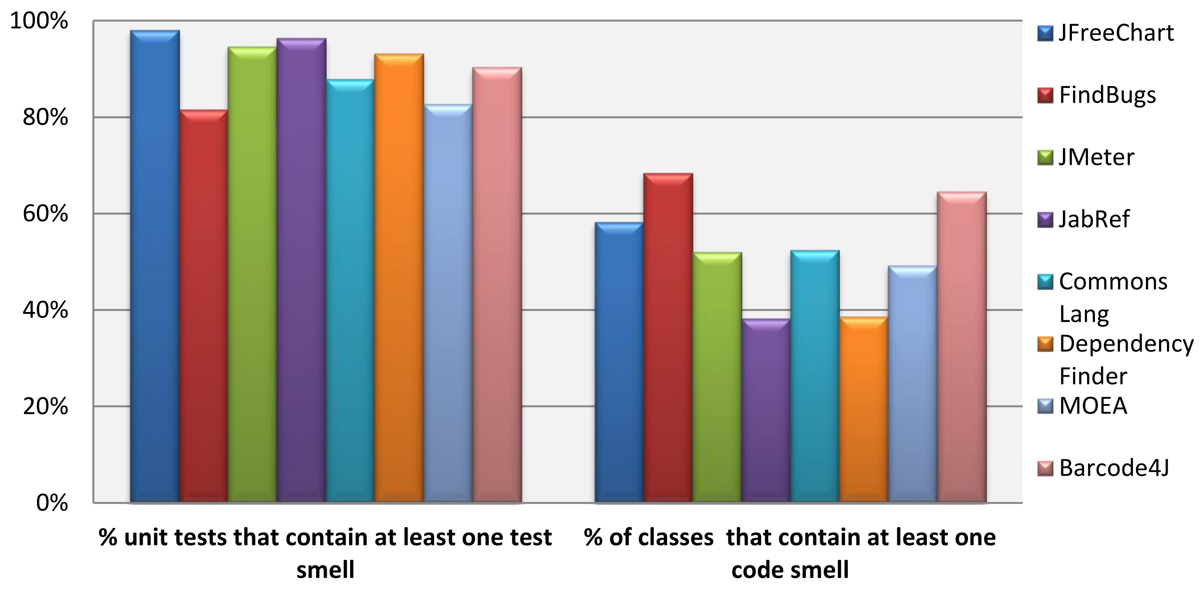


Figure 30. Percentage of unit tests and production classes that contain smells

7.3 Results

The results first address the distribution of smells across all eight OSS. *Assertion Roulette* was found to be the most frequently occurring test smell in all systems, as it makes up between 34% and 48% of the total number of test smell types in seven of the eight systems (see Figure 28). Its occurrence in JFreeChart was an even higher exception, as *Assertion Roulette* accounts for almost 71% of the test smells captured in that system. The *Duplicated Code* test smell was the second most commonly distributed smell in all eight systems. On the other hand, *Indirect Test* and *Lazy Test* appear to be the smells that are least likely to occur. *Lazy Test* and *Indirect Test* both form less than 5% of the total number of smells in all studied systems. For code smells, *Duplicated Code* and *Brain Class* appear to be the most commonly occurring smells in all eight systems (Figure 29), while *Message Chain* and *Tradition Breaker* are the least widely distributed code smells.

Table 27. Number of unit tests that contain different test smells

Test Smell	JFreeChart	FindBugs	JMeter	Commons Lang	JabRef	Dependency Finder	MOEA	Barco-de4J
Assertion-free	9	2	19	31	5	15	8	4
Assertion Roulette	347	21	63	85	51	172	140	25
Sensitive Equality	6	3	14	58	5	11	2	16
Mystery Test	0	8	7	5	10	29	23	0
Indirect Test	0	2	7	2	1	22	3	2
General Fixture	28	11	23	16	9	112	27	0
Duplicated Code	59	5	35	36	17	85	66	7
Eager Test	33	3	11	14	8	30	19	10
Lazy Test	9	2	4	5	2	0	4	1
Total	**491**	**57**	**183**	**252**	**108**	**476**	**292**	**65**

In the analyses reported in the remainder of this section we interpreted there to be a significant relationship between two variables if there is statistically significant evidence of such a relationship in at least five of the eight examined systems (and where details of the definition of statistical significance were explained in Chapter

3, Section 3.2.2). The discussion and the consideration of the implications of the findings follow later in Section 7.4.

The presentation of the results related to the RQs are divided into three parts: the first part shows the results related to $RQ_{7.1}$ through $RQ_{7.5}$, which investigate a number of *bivariate* correlations between smell types and software artefacts characteristics of size and complexity. The second part presents all results related to $RQ_{7.6}$ and $RQ_{7.7}$, which look at the relationship between individual smells and unit test size. The third part shows the results related to $RQ_{7.8}$ and $RQ_{7.9}$, which investigate the co-occurrence of test and code smells.

Table 28. Number of production classes that contain different code smells

Code Smell	JFreeChart	FindBugs	JMeter	Commons Lang	JabRef	Dependency Finder	MOEA	Barco-de4J
Feature Envy	76	3	15	2	1	20	18	2
Large Class	54	6	13	33	4	29	20	6
Type Checking	5	6	4	2	0	6	9	0
Brain Class	63	23	8	33	9	36	29	7
Duplicated Code	122	17	23	40	16	33	48	15
Schizophre-nic Class	3	1	0	1	0	6	3	0
Data Class	10	0	1	3	0	4	4	0
Data Clumps	59	0	0	2	2	0	3	1
Tradition Breaker	0	1	1	0	0	1	0	0
Message Chain	0	0	1	0	0	0	0	0
Total	**392**	**57**	**66**	**116**	**32**	**135**	**134**	**31**

Relationships between smell types and software artefacts

For $RQ_{7.1}$, Table 29 reports the results of the correlation analyses between size and the number of test smell types in a unit test. Results of the correlation analyses between size and the number of code smell types in the corresponding production classes are shown in Table 30 (RQ7.2). Note that all medium and strong correlation coefficient values in Tables 29 through 32 are reported in **Bold**[40].

Table 29. Spearman's *ρ* correlation between test smell types and unit tests' size metrics

Metrics		Number of Test Smell Types							
		JFreeChart	FindBugs	JMeter	JabRef	Commons Lang	Dependency Finder	MOEA	Barcode4J
TLOC	*ρ*	**.33**	**.59**	**.74**	**.72**	**.65**	**.63**	**.54**	**.66**
	p	**.00**	**.00**	**.00**	**.00**	**.00**	**.00**	**.00**	**.00**
NTC	*ρ*	.26	**.48**	**.71**	**.73**	**.57**	**.49**	.26	**.65**
	p	.00	**.00**	**.00**	**.00**	**.00**	**.00**	.00	**.00**

For the relationship between test smell types and size (RQ7.1, Table 29), it was found that TLOC in a unit test has a significant positive correlation with the number of test smell types present in a unit test in all eight systems studied. The relationship between TLOC and the number of test smell types present in a unit test was typically high (JFreeChart was the only exception here, where the strength of the relationship is medium). In addition, NTC was found to have a significant correlation with the number of test smell types in a unit test. However, the correlation between NTC and the number of test smell types is weaker than the correlation between TLOC and the number of test smell types in a unit test. Evidence indicated a high correlation between NTC and the number of test smell types in four systems (JMeter, JabRef, Commons Lang and Barcode4J), a medium correlation in FindBugs and Dependency Finder, and a low correlation in JFreeChart and MOEA. Therefore, *H7.1* is accepted: *there is a significant positive relationship between the size of a unit test and the number of test smell types in that unit test*. The larger the unit test, the higher the number of test smell types in the unit tests.

[40] Note that most of the non-bold entries in Tables 7-10 are still significant but their correlations are rather weak.

Table 30. Spearman's ρ correlation between code smell types and unit tests' size metrics

Metrics		Number of Code Smell Types							
		JFreeChart	FindBugs	JMeter	JabRef	Commons Lang	Dependency Finder	MOEA	Barco-de4J
TLOC	ρ	.26	.13	.27	**.45**	**.50**	**.37**	**.36**	**.53**
	p	.00	.22	.02	**.00**	**.00**	**.00**	**.00**	**.00**
NTC	ρ	.18	.11	.28	**.44**	**.40**	.25	.24	.32
	p	.00	.25	.02	**.00**	**.00**	.00	.00	.08

In regard to $RQ_{7.2}$, the degree of association between the number of code smell types in a production class and the size of its corresponding unit test was investigated. As shown in Table 30, it was found that there was a significant correlation between the number of code smell types in a production class and TLOC in its corresponding unit test. High correlations between the number of code smell types and TLOC were found in Apache Commons Lang and Barcode4J. Medium correlations were found in JabRef, Dependency Finder and MOEA, and a low correlation was found in JFreeChart and JMeter. Overall the correlation between the number of code smell types in a production class and TLOC was found to be generally stronger than the correlation between code smells and NTC. For the latter relationship, it was found that there were medium correlations in JabRef and Apache Commons Lang, whereas low correlations were noted in four systems (i.e., JFreeChart, JMeter, Dependency Finder and MOEA). No significant correlations for NTC were found in FindBugs and Barcode4J. Therefore, $H_{7.2}$ is also accepted: *there is a significant positive relationship between the size of the unit test and the number of code smell types in the associated production class*. That is, the higher the number of code smell types in a production class, the larger the corresponding unit test.

Table 31 reports all of the statistically significant correlations between the number of test smell types and the size and complexity of the corresponding production code ($RQ_{7.3}$ and $RQ_{7.4}$). Note that no significant correlations were found in

FindBugs, while JMeter showed a significant correlation only with LOC. The size of the production class was found to be positively correlated with the number of test smell types in its corresponding unit test ($RQ_{7.3}$). LOC was also shown to have a high positive correlation with the number of test smell types in JFreeChart, JabRef, Commons Lang and Barcode4J, a medium correlation in Dependency Finder and MOEA, and a low correlation in JMeter. Number of Methods (NOM) per class was also shown to be correlated with the number of test smell types in the corresponding unit test in six systems (although this correlation was slightly weaker than the correlation between the number of test smell types and LOC). The strength of the correlation was high in JabRef, Commons Lang and Barcode4J, medium in JFreeChart and low in Dependency Finder and MOEA. We therefore accept $H_{7.3}$: *there is a significant positive relationship between the number of test smell types present in a unit test and the size of the corresponding production class*. That is, the number of code smell types in a unit test increases when the size of the production class increases, or vice versa.

Table 31. Spearman's *ρ* correlation between test smell types and size and complexity of the corresponding production class

Metrics		**Number of Test Smell Types**							
		JFreeChart	**FindBugs**	**JMeter**	**JabRef**	**Commons Lang**	**Dependency Finder**	**MOEA**	**Barco-de4J**
LOC	*ρ*	**.51**	-.11	.26	**.58**	**.56**	**.35**	**.30**	**.54**
	p	**.00**	.26	.02	**.00**	**.00**	**.00**	**.00**	**.00**
NOM	*ρ*	**.44**	.03	.18	**.68**	**.50**	.25	.28	**.50**
	p	**.00**	.42	.12	.00	**.00**	.00	.00	**.00**
CC	*ρ*	.24	-.25	.20	.22	.19	**.38**	.24	.02
	p	.00	.07	.09	.05	.02	**.00**	.00	.94

Turning to $RQ_{7.4}$, class complexity (which is represented here by the CC metric) was found to be significantly correlated with the number of test smell types in the corresponding unit test (Table 31, last row), in five of the eight systems examined.

However, although the correlations were found to be significant, the strength of the correlations were not high: a medium correlation between the number of test smell types and CC was found in Dependency Finder, whereas a low correlation was found in JFreeChart, JabRef, Commons Lang and MOEA (and we did not find any significant correlations in FindBugs, JMeter and Barcode4J). On balance *$H_{7.4}$* is therefore also accepted: *there is a significant positive relationship between the number of test smell types present in a unit test and the complexity of the corresponding production class.* More complex classes are *generally* more likely to have higher numbers of test smell types in their associated unit tests, and vice versa.

Table 32. Spearman's *ρ* correlation between test and code smell types

		Number of Test Smell Types							
		JFreeChart	**FindBugs**	**JMeter**	**JabRef**	**Commons Lang**	**Dependency Finder**	**MOEA**	**Barco-de4J**
Number of Code Smell Types	*ρ*	**.51**	.17	**.37**	**.46**	**.50**	.22	**.39**	**.53**
	p	**.00**	.15	**.00**	**.00**	**.00**	.00	**.00**	**.00**

One of the most interesting findings that emerged from the analysis was the significant positive correlation between the number of test smell types in a unit test and the number of code smell types in the corresponding production class ($RQ_{7.5}$). As shown in Table 32, there are significantly high correlations between the number of test and code smell types in three systems (i.e. JFreeChart, Commons Lang and Barcode4J), medium correlations in three further systems (i.e., JabRef, JMeter and MOEA) and a low correlation in Dependency Finder. Again, no significant correlation was found in FindBugs.

We therefore accept the hypothesis for this question, *$H_{7.5}$: there is a significant positive relationship between the total number of test smell types in a unit test and the total number of code smell types in the corresponding production class.* Whenever the number of code smell types in a production class increases, the number of test smell types in the associated unit tests also increases, and vice versa.

Table 33. Mann-Whitney U test results with effect size for the relationship between test size metrics and test smell types

Smells	Metric		JFreeChart	FindBugs	JMeter	JabRef	Commons Lang	Dependency Finder	MOEA	Barcode4J
Assertion-free	TLOC	p	.39	1.00	.15	1.00	.13	.66	.68	.68
		ez								
	NTC	p	1.00	.98	.06	.48	.14	.29	.21	1.00
		ez								
Assertion Roulette	TLOC	p	.42	1.00	**.01***	.38	**.00***	.01*	.00*	.52
		ez			**0.4**		**0.3**	0.2	0.2	
	NTC	p	.50	.92	**.01***	1.00	**.02***	.31	.78	.35
		ez			**0.4**		**0.3**			
Sensitive Equality	TLOC	p	1.00	.675	**.05***	**.02***	.14	1.00	.68	.10
		ez			**0.3**	**0.4**				
	NTC	p	.76	1.00	**.02***	**.02***	.13	.86	.67	.12
		ez			**0.3**	**0.4**				
Mystery Guest	TLOC	p	**----**	.33	.62	.26	.04*	.04*	**.00***	-----
		ez					0.2	0.2	**0.3**	
	NTC	p	**----**	1.00	.60	.46	.18	.00*	.02*	-----
		ez						0.2	0.2	
Indirect Test	TLOC	p	**----**	1.00	.16	1.00	.69	.01*	.10	1.00
		ez						0.2		
	NTC	p	**----**	.63	.24	1.00	.89	.25	.53	.64
		ez								
General Fixture	TLOC	p	.91	**.02***	**.00***	.08	.04*	**.00***	**.00***	-----
		ez	-----	**0.5**	**0.5**		.2	**0.3**	**.4**	
	NTC	p	.25	**.01***	**.00***	.32	-----	.02*	**.00***	-----
		ez		**0.5**	**0.5**			0.2	**0.3**	
Eager Test	TLOC	p	.00*	.29	.60	**.05***	**.00***	**.56**	.62	.23
		ez	0.2			**0.4**	**0.4**			
	NTC	p	.01*	1.00	.48	**.01***	**.00***	.21	1.00	.96
		ez	0.2			**0.5**	**0.3**			
Lazy Test	TLOC	p	1.00	.90	.84	.82	.17	**----**	.58	1.00
		ez								
	NTC	p	1.00	.88	.42	.83	.19	----	1.00	.91
		ez								
Duplicated Code	TLOC	p	**.00***	**.01***	**.00***	**.00***	**.00***	**.00***	**.00***	**.01***
		ez	**0.4**	**0.5**	**0.6**	**0.5**	**0.4**	**0.6**	**0.3**	**0.6**
	NTC	p	**.00***	**.04***	**.00***	**.00***	**.00***	**.00***	.72	**.00***
		ez	**0.3**	**0.5**	**0.5**	**0.5**	**0.3**	**0.5**		**0.7**

Relationships between individual smell types and unit tests size

The next component of the experiment investigates which particular smells are more closely associated with unit test size ($RQ_{7.6}$ and $RQ_{7.7}$). Tables 33 and 34 report the values of the Mann-Whitney U tests and effect size for the relationships between the test's size metrics and test and code smell types, respectively. All significant values are marked with an asterisk (*) and all medium and high effect sizes are shown in **Bold**. Results that are significant for *five or more* of the eight systems are highlighted.

As shown in Table 33, tests' *Duplicated Code* smell type was found to have a significant correlation with TLOC in all of the systems studied, and with NTC in seven of the eight systems. Among those, five systems are shown to have a high effect size with TLOC and NTC, in FindBugs, JMeter, JabRef, Dependency Finder and Barcode4J, and medium effect size in JFreeChart, Commons Lang and MOEA[41]. *General Fixture* is another smell type that was found to be closely associated with size. *General Fixture* and TLOC are significantly correlated in five systems (with high effect size in FindBugs and JMeter, medium effect size in Dependency Finder and MOEA, and low effect size in Commons Lang). The relationship between *General Fixture* and NTC was evident only in four of the eight systems. On the balance of evidence, $H_{7.6}$ is accepted for two smell types: *General Fixture* and *Duplicated Code*; that is, *there is a significant difference in the size of unit tests that contain the General Fixture or Duplicated Code smell types and unit tests that do not contain these smell types*.

It should be noted here that there was no expectation of finding any statistically significant results for the *Message Chain* and *Tradition Breaker* code smells because of the very limited occurrence of these smells (see Figure 29 and Table 28). *Message Chain* appeared only once in JMeter, while *Tradition Breaker* appeared only once in

[41] Note that *Duplicated Code* was not significantly correlated with NTC in MOEA.

three systems i.e., FindBugs, JMeter and Dependency Finder. Therefore, no correlation analysis was performed on these individual smells, and they are excluded from any further analysis.

Table 34. Mann-Whitney U test results with effect size for the relationship between test size metrics and code smell types

Smells	Metric		JFreeChart	FindBugs	JMeter	JabRef	Commons Lang	Dependen-cy Finder	MOEA	Barco-de4J
Feature Envy	TLOC	p	.00*	1.00	.35	.12	.60	.05*	.12	.75
		ez	0.2					0.2		
	NTC	p	.56	1.00	.35	.32	.35	.68	.86	.68
		ez								
Large Class	TLOC	p	.19	1.00	.54	.16	**.00***	**.48**	**.06**	.39
		ez					**0.4**			
	NTC	p	.55	1.00	.70	.16	**.00***	.58	.27	.49
		ez					**0.4**			
Type Checking	TLOC	p	.30	1.00	.64	----	1.00	.08	.02*	----
		ez							0.2	
	NTC	p	1.00	1.00	**.91**	----	.67	.75	.31	----
		ez								
Brain Class	TLOC	p	.02*	.42	**.05***	.08	**.02***	**.00***	**.00***	**.01***
		ez	0.1		**0.3**		**0.3**	**0.3**	**0.3**	**0.6**
	NTC	p	.95	1.00	**.05***	.12	.14	.07	**.00***	.13
		ez			**0.3**				**0.3**	
Duplicated Code	TLOC	p	**.00***	.80	1.00	**.01***	**.00***	.03*	.28	.14
		ez	**0.3**			**0.4**	**0.4**	0.2		
	NTC	p	.00*	1.00	1.00	**.01***	**.01***	.13	1.00	.38
		ez	0.2			**0.4**	**0.3**			
Schizophre nic Class	TLOC	p	.27	**.63**	----	----	**1.00**	**.99**	**.39**	----
		ez								
	NTC	p	.51	1.00	----	----	1.00	**.75**	**.78**	----
		ez								
Data Class	TLOC	p	.02*	----	1.00	----	**1.00**	**.87**	.53	----
		ez	0.2							
	NTC	p	.00*	----	1.00	----	**1.00**	.40	.48	----
		ez	0.2							
Data Clumps	TLOC	p	.00*	----	----	.18	0.91	----	.61	.44
		ez	0.2							
	NTC	p	**.00***	----	----	.12	1.00	----	.91	.68
		ez	**0.3**							

Table 34 reports the results of the Mann-Whitney U tests and effect size for the correlations between size metrics and code smell types ($RQ_{7.7}$). These results indicate that the *Brain Class* smell type was more closely associated with unit test size metrics than other code smell types. The relationship between *Brain Class* and TLOC was evident in Barcode4J (with high effect size), JMeter, Commons Lang, Dependency Finder and MOEA (medium effect size) and in JFreeChart (low effect size). In contrast, *Brain Class* is significantly correlated with NTC in only two of the eight systems (with medium effect size in Dependency Finder and MOEA). *Duplicated Code* was found to be significantly associated with TLOC in four systems: JFreeChart, JabRef and Commons Lang (with medium effect size), and Dependency Finder (low effect size). Given these results, $H_{7.7}$ is accepted only for *Brain Class* smell type: *There is a significant difference in the size of unit tests that contain Brain Class code smell type in the associated production class and those that do not contain this smell.*

Smells co-occurrence

We now turn our attention to examining the co-occurrence of code and test smell types ($RQ_{7.8}$). Table 35 presents all significant correlations between code and test smell types across all eight studied systems. All of the reported correlations here are positive in direction. As explained earlier in this section, we employed a threshold of 5 or more systems to indicate the presence of a significant correlation between any two investigated variables. However, we also report significant correlation values that fall below this threshold (i.e., that occur at least in three of the eight systems) mainly to provide a general view of other potentially important correlations that appear in all systems (and this approach also applies to Table 36). Full results of the correlation analysis between test and code smell types for all eight systems are provided in Appendix E.

Table 35. Results of the *phi* (*φ*) correlation coefficient analysis between test and code smells

(Shading denotes finding significant co-occurrence in 4 or more systems. Red bold is used to highlight High (H) strength correlations, Blue *italic* to highlight Medium (M) and Green to highlight Low (L) correlations)

Test Smells/ Code Smells	Assertion-free	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	*(M) JabRef* ϕ= 0.43, p = 0.00 (L) JFreeChart ϕ=0.22 , p =0.00 (L) Commons Lang ϕ= 0.22, p = 0.01		*(M) JFreeChart* ϕ=0.35, p= 0.00 *(M) JabRef* ϕ= 0.33, p= 0.01 *(M) MOEA* (ϕ=0.37, p= 0.00 *(M) Barcode4J* ϕ= 0.38, p= 0.03	**(H) Barcode4J** **ϕ=0.70, p= 0.00** *(M) JMeter* ϕ=0.33, p= 0.01 (L) JFreeChart ϕ=0.14, p= 0.01	
Large Class			*(M) JMeter* ϕ= 0.31, p= 0.01 *(M) Commons Lang* ϕ= 0.31, p= 0.00 *(M) MOEA* (ϕ= 0.35, p= 0.00 (L) JFreeChart ϕ=0.27 , p= 0.00 (L) JabRef ϕ=0 .28, p=0.04		
Brain Class	(L) JFreeChart ϕ=0.21, p= 0.00 (L) Commons Lang ϕ= 0.20, p= 0.03 (L) MOEA ϕ= 0.28, p= 0.00	*(M) JFreeChart* ϕ=0.41, p= 0.00 *(M) JMeter* ϕ=0.33, p= 0.00 (L) MOEA ϕ=0.21, p= 0.00	(M) JFreeChart ϕ= 0.34, p= 0.00 *(M) JabRef* ϕ= 0.38, p= 0.01 *(M) Commons Lang* ϕ= 0.42, p= 0.00 *(M) Barcode4J* ϕ= 0.45, p= 0.01 (L) JMeter ϕ= 0.22, p= 0.05 (L) MOEA ϕ= 0.20, p= 0.00	*(M) JMeter* ϕ=0.30, p=0.01 *(M) MOEA* ϕ=0.35, p=0.00 (L) JFreeChart ϕ=0.16, p=0.00	*(M) FindBugs* ϕ=0.31, p= 0.05 (L) JFreeChart ϕ=0.17, p= 0.00 (L) Commons Lang ϕ=0.22, p= 0.02 (L) Dependency Finder ϕ=0.20, p= 0.00
Duplicated Code			(L) Commons Lang ϕ=0.198 , p= 0.03 (L) JFreeChart ϕ= 0.24 , p= 0.00 (L) MOEA ϕ= 0.22, p= 0.00		**(H) JabRef** **ϕ=0.70, p= 0.00** **(H) Commons Lang** **ϕ=0.51, p= 0.00** *(M) JFreeChart* ϕ=0.27, p= 0.00 (L) Dependency Finder ϕ=0.14, p= 0.04

As shown in Table 35, the *Eager Test* smell type was found to be significantly correlated with *Brain Class* (in six systems) and with *Large Class* (in five systems). *Eager Test* was found to be significantly correlated with *Brain Class* in JFreeChart, JabRef, Commons Lang and Barcode4J (medium correlations) and JMeter and MOEA (a low correlation). A significant correlation between *Eager Test* and *Large Class* is evident in JMeter, Commons Lang and MOEA (medium), and JFreeChart and JabRef (both low). *$H_{7.8}$*, therefore, is also accepted – *there is a significant relationship between the co-occurrence of individual test smell types and individual code smell types* – for the following pairs of test and code smell types: *Eager Test* and *Brain Class*, and *Eager Test* and *Large Class*.

There are three other correlations that occur in only four systems but that are worth noting (Table 35). Particularly, test smell *Duplicated Code* is significantly correlated with code smell *Duplicated Code*. There is a high correlation between *Duplicated Code* in production and test code in JabRef and Commons Lang, medium in JFreeChart and low in Dependency Finder. In addition, a significant medium correlation between the *Eager Test* and *Feature Envy* smell types was found in JFreeChart, JabRef, MOEA and Barcode4J. Similarly, *Brain Class* and test *Duplicated Code* smell types are shown to be correlated in the JFreeChart, FindBugs, Common Lang and Dependency Finder systems.

Table 36. Results of the *phi* (*φ*) correlation coefficient analysis between test smell types (Blue *italic* to highlight Medium (M) and Green to highlight low (L) correlations)

Test Smells	Assertion Roulette	General Fixture	Eager Test

Duplicated Code	*(M): FindBugs*
	ϕ= 0.35, p= 0.03
	(L): Commons Lang
	ϕ= 0.20 , p= 0.02
	(L): Dependency Finder
	ϕ= 0.15, p= 0.03
	(L): MOEA
	ϕ= 0.17, p= 0.02
		(L): JFreeChart
		ϕ= 0.18, p= 0.00
		(L): JMeter
		ϕ= 0.25, p= 0.03
		(L): Commons Lang
		ϕ= 0.18, p= 0.05
			(L): JFreeChart
			ϕ= 0.17, p= 0.00
			(L): MOEA
			ϕ= 0.17, p= 0.02
			(L): Commons Lang
			ϕ= 0.28, p= 0.00

In considering $RQ_{7.9}$, Table 36 presents the results of the correlation analysis among test smells. As in Table 35, only significant correlations between test smell types that are evident in at least three systems are reported. Detailed results of the correlations between test smell themselves are available in Appendix F. As shown in Table 36, it is found that there are three different test smell types that co-occur with the *Duplicated Code* test smell in the same unit test. There is a significant relationship, although not strong, between *Assertion Roulette* and test *Duplicated Code*. This relationship was found in four systems. These two smell types had some tendency to co-occur in a unit test in FindBugs (medium correlation), Commons Lang, Dependency Finder and MOEA (low correlation). *Duplicated Code* also co-occurs (in three systems) with *General Fixture* (in JFreeChart, JMeter and Commons Lang) and *Eager Test* (in JFreeChart, MOEA and Commons Lang). However, these correlations are all shown to be low.

Based on the results presented above, $H_{7.9}$ is rejected: we found no evidence that *there is a significant relationship between the co-occurrence of individual test smell types*, as we did not find evidence of any correlation among smells consistently across five or more systems.

However, it is also worth noting – though as a separate observation that should not influence the acceptance or rejection of the examined hypothesis – that there are other potentially interesting correlations among test smell types that occur only in one or two systems. For example, a high negative correlation between *Assertion-free* and *Assertion Roulette* was found in Barcode4J. Such a finding seems reasonable as one would expect that there would be a negative correlation between the two smells given that *Assertion Roulette* and *Assertion-free* are opposite to each other (i.e., *Assertion Roulette* appears where there are multiple assertion statements in a test case, whereas *Assertion-free* occurs when there are no assertion statements in a test case).

The following section provides a discussion on the abovementioned results.

7.4 Discussion

In the following we discuss our results. We divide the discussion into three parts: the first part provides a discussion on the bivariate correlations between smell types and artefact size and complexity (RQ$_{7.1}$-RQ$_{7.5}$), followed by an analysis of the effect of size. The second part presents a discussion on the relationships between individual smell types and unit test size (RQ$_{7.6}$ and RQ$_{7.7}$). The final part provides a discussion on the co-occurrence of test and code smells (RQ$_{7.8}$ and RQ$_{7.9}$).

Relationships between smell types and software artefacts

The findings of this experiment show that an increase in the size of both a production class and a unit test will likely be mirrored by an increase in the number of test smell types in that unit test: the size of both production classes and unit tests was found to be significantly associated with the number of test smell types. This result is consistent with findings from prior research that showed that the size of a class generally constitutes a useful indicator of problems or issues in code (Yamashita and Counsell, 2013). In their system-level analysis Bavota et al. (2014) also found that the number of test smell types in a system is related to the size of that system, and other prior research found evidence of a significant relationship between the size of production classes and their associated unit tests (Bruntink and van Deursen, 2006). To summarise these particular findings, a large production class will typically require a large unit test, and a large unit test is more likely to contain test smells.

It was also observed that the number of test smell types in a unit test increases when the complexity of the associated production class increases: more complex classes are likely to have higher numbers of test smell types in their associated unit tests. Relationships between complexity and unit test size have been found elsewhere. For instance, complexity-related metrics, such as cohesion, and Response for a Class (RFC), have been found to be significantly correlated with the size of unit tests (Bruntink and van Deursen, 2006, Badri et al., 2010, Zhou et al., 2012). Dynamic coupling is also associated with unit test size, as was demonstrated in Chapter 5. We therefore find the correlation between class complexity and the number of test smell types in the associated unit test to be plausible. Developers may attempt to test complex modules by writing large unit tests, which therefore increase the chances of introducing smells in the unit test (as shown in $RQ_{7.1}$).

It is important at this juncture to reiterate that this research did not investigate the causation (i.e., direction) of these relationships[42]. It is therefore not possible to say if any of the variables considered *directly* influences the others. However, causation of a relationship could be identified if testing strategy information were obtained from a project. For example, if a project was developed following a traditional test-last approach, then, since production classes are developed first, it is not possible for the production classes to be influenced by the test code. It may be possible in some situations to infer that test code was influenced by production code and its characteristics. However, it is also possible for both production and test code to reflect other factors, such as the complexity of the problem being solved. Similarly, in a test-first method (such as Test Driven Development), it is impossible for production code to influence test code, since test code is developed first. This limitation applies to all other similar relationships discussed here (especially for $RQ_{7.3}$, $RQ_{7.4}$ and $RQ_{7.5}$), and is an aspect of analysis that should be addressed in future research.

That said, it is also important to note that a recent empirical study explained that the majority of OSS projects and developers follow a test-last approach, i.e., test code is developed after the production code (Beller et al., 2015). The authors empirically investigated developers' testing practices by studying 416 developers from 460 projects. Their study found that TDD is not widely practiced – in only 12% of the projects that claimed to follow TDD did the developers actually follow

[42] See Sections 3.5.1 and 7.5 for more details.

TDD practices (Beller et al., 2015). Moreover, even when developers claim that they practice TDD, it was found that they do not follow its practices strictly (i.e., tests are still being written after production classes are developed). In light of these findings we could cautiously assert that the direction of the relationships identified in this chapter are likely to follow a [production class ➔ unit test] direction.

The effect of size

In this section, we statistically examined the possible confounding effect of size on the investigated correlations between test and code smell types in $RQ_{7.1}$ through $RQ_{7.5}$. We measured the indirect effect of size (the confounding variable in this case) on all causal relationships between variables using the Bootstrapping mediation method. Mediation and confounding effects have been proven to be statistically equivalent (MacKinnon et al., 2000), and therefore methods of mediation effect may be used to investigate possible confounding effects (Preacher and Hayes, 2008). Bootstrapping is a non-parametric approach to effect-size estimation and hypothesis testing that makes no assumptions about the distribution of the data or the sampling distribution of the statistic (Preacher and Hayes, 2004). This method has been shown to be more powerful than other similar approaches, such as the Sobel test, and therefore has been recommended to be used over other mediation methods (MacKinnon et al., 2004, Hayes, 2009). Bootstrapping and other mediation analysis methods are explained in Preacher and Hayes (2008) and Hayes and Preacher (2014). In this method, we first 'bootstrap' the data by taking a large number of samples of size n (where n is the original sample size) from the data, sampling with replacement, and computing the indirect effect between variables x and y through the presence of the mediator variable m in each sample. As explained by Preacher and Hayes (2004, 2008), let's say we have 1000 bootstrap samples, the point estimate of variables x and y is simply the mean of the two variables computed over the 1000 samples, and the estimated standard error is the standard deviation of the 1000 *xy* estimates. To derive a 95% confidence interval, the elements of the vector of 1000 estimates of *xy* are sorted from low to high. The Lower Limit (LL) (also known as the lower boundary) of the confidence interval is defined as the 25th score in this sorted distribution, and the Upper Limit (UL) (also known as the upper boundary) is defined as the 976th score in the distribution. If *zero* (0) falls between the resulting confidence interval values of LL and UL then it is concluded (with 95% confidence) that there is no significant mediation (or confounding) effect: that is, the indirect effect is *non-significant*.

As explained above, the goal is to examine the confounding effect of size, and therefore we use our two key size metrics (LOC and TLOC) as mediators. The procedure was applied to the data obtained from all 8 OSS. As before, we use a threshold of 5 or more systems to indicate the presence of a confounder (mediator) between any two investigated variables. For $RQ_{7.1}$, $RQ_{7.2}$, $RQ_{7.4}$ and $RQ_{7.5}$ we examined the indirect effect of LOC; for $RQ_{7.3}$, we considered TLOC as a potential mediator. Detailed results of this procedure are shown in Appendix G. This procedure was carried out using the MEDIATE[43] SPSS macro. We used a confidence interval of 95% and employed 5000 bootstrap resamples, as has been recommended in the literature (Preacher and Hayes, 2008).

The results of the mediation analysis confirm that LOC and TLOC do not appear to have a significant indirect effect on the relationship between the number of test smell types and unit test's size ($RQ_{7.1}$) or class size ($RQ_{7.3}$) in seven of the examined systems. A similar outcome applies to the relationship between the number of code smell types and unit test size ($RQ_{7.2}$), as there is no evidence that class size has an indirect effect on this relationship in five of the eight systems examined.

[43] http://www.afhayes.com/spss-sas-and-mplus-macros-and-code.html

Similarly, the indirect effect of LOC between the number of test smell types and class complexity ($RQ_{7.4}$) is not evident in five of the eight systems. The same result applies to the indirect effect of LOC between the number of test smell types and code smell types ($RQ_{7.5}$) – as there is no evidence of a significant indirect effect in seven of the eight systems.

Based on the findings of our bootstrap mediation analysis, we can state (with confidence of 95%) that size does not have a significant confounding effect on the relationships investigated in this chapter.

Relationships between individual smell types and unit tests size

By looking more deeply into the relationships between size and the incidence of individual smell types, there are specific smell types that are shown to be more strongly associated with size than others. Among all of the test smells studied, *General Fixture* and *Duplicated Code* are more closely related to the size of the unit test than other smell types. As the size of the unit test increases (i.e., in terms of both TLOC and NTC), the possibility of developing *General Fixture* and *Duplicated Code* test smell types also increases (or vice versa). The result regarding *General Fixture* is also in line with the findings of Greiler et al. (2013b), which explained how other *test fixture* related smells are often correlated with the number of test cases.

Turning our attention to code smells, *Brain Class* has been shown to be more strongly associated with the size of the unit test compared to the other code smell types examined. One could therefore expect that a class with a *Brain Class* smell and/or *Duplicated Code* will have a larger unit test. *Brain Class* represents an "excess of responsibility" of a class, whereas *Duplicated Code* represents "unnecessary code" that should be removed. This would indicate that a class with a relatively higher number of responsibilities is likely to have a larger unit test. Previous work by Sabane et al. (2013) indicated that classes with smells related to an "excess of responsibility", such as *Blob Class,* require significantly higher numbers of test cases than other classes.

Smell co-occurrence

A further novel finding of this work is that relating to the co-occurrence of test and code smells. Based on the preceding analysis it appears that some test and code smell types are more likely to co-occur. For instance, *Eager Test* is the smell type most significantly associated with other code smell types, and particularly *Brain Class* and *Large Class* code smell types. To provide greater context for this discussion the smells classification and taxonomy developed by Mäntylä et al. (2003) is used.

The relationship between *Eager Test* and *Brain Class* was evident in six of the eight systems examined. As explained above, *Brain Class,* which belongs to the "complexity" group of code smells (Mäntylä et al., 2003), represents an "excess of responsibility" class. To test a complex class, such as a *Brain Class,* a unit test might attempt to test multiple methods of the same class under test, leading to possible interdependencies between methods within the same class. In doing so, a test might require the involvement (e.g., invoke) of several other methods that are related or connected to the target method under test, which can lead to the introduction of the *Eager Test* smell (where a test has a method that uses more than one method of the tested class).

Much the same interpretation can be also drawn for the relationship between *Eager Test* and *Large Class*. A *Large Class* smell is an indicator that a class has at least one large method. Such methods can be also linked (connected) with other methods within the same class. One possible scenario is that, when writing a test case to test a particular method that is large in size (given the possible interdependency between the large method and other methods in the same class), a developer/tester might be required to test multiple methods in the same class at once. It is also possible that methods within a class are sometimes required to access parameters or operators that belong to those large methods within the same class. When attempting to test such methods, a test is required to make use (i.e., invoke or retrieve data from) other methods of the same class under test.

The relationship between test and production code duplication (i.e., *Duplicated Code*) is another potentially interesting finding, although it was not consistent among all systems considered (as it appeared in four of the eight systems)[44]. This relationship suggests that duplication in code can also be a sign of possible code duplication in the test code, and vice versa.

The high diffusion of *Assertion Roulette* leads us to expect that this smell type is more associated with the size of the unit test than other test smells (i.e., unit tests with *Assertion Roulette* are larger compared to unit tests without this smell). The argument here is that a large unit test has more test cases (given that there is a positive relationship between the number of test cases and the size of unit tests). The increase in the number of test cases (i.e., NTC) means that there are more Assertions in the unit test. We therefore expected that the more Assertions in a unit test, the higher the probability of introducing *Assertion Roulette.* However, the findings of this study show that this is not case for at least half of the studied systems. On the other hand, one would expect that *Assertion Roulette* would be correlated with some of the code smell types that are related to size (such as *Large Class*) or other widely distributed code smells, as *Assertion Roulette* is shown to be the most common test smell in OSS projects. Again, this is was not the case here as *Assertion Roulette* was not found to co-occur with any other code smells.

In studying the co-occurrence of test smell types themselves, there were no obvious correlations among different individual test smell types. There was no statistical evidence that any two test smell types co-occurred together in more than

[44] As shown in Chapter 7, we use a threshold of five systems or more to confirm significant associations between any two smells. Any correlations that occurred in fewer than five systems would not be considered as a significant correlation between the variables.

four systems. The only relationship perhaps worth noting is that between *Assertion Roulette* and test *Duplicated Code*, although this was found in just four of the eight systems. This overall result indicating a general lack of test smell co-occurrence is in contrast to the previous analyses of Bavota et al. (2014), which found that most test smells co-occur with *Assertion Roulette.* These authors also found that *Lazy Test* and *Eager Test* smells co-occur on several occasions. However, the method these authors used to examine co-occurrence considers the direction of the relationship (i.e., the co-occurrence of A and B may be different from the co-occurrence of B and A). This led to a number of correlations that are not symmetrical. For example, *Assertion Roulette* was shown to co-occur with *Lazy Test* in 83% of the unit tests, but if we swap these variables (i.e., the co-occurrence of *Lazy Test* and *Assertion Roulette*) the percentage drops to only 3%. In order to avoid such non-symmetrical outcomes we used an alternative method of analysis (i.e., *Phi Correlation Coefficient Test*) that assesses co-occurrence based on the strength of the association (i.e., significance level and the degree of association) between all individual pairs of test smell types.

Threats to the validity to this experiment are presented next.

7.5 Threats to Validity

There are a number of threats to the validity of our work, as acknowledged in the following. Note that the nature of some of these threats is explained more fully in Section 3.5.

Possible Confounding Effect of Size

As explained in Section 3.5.1., some of the correlations identified between variables might be influenced by confounding factors (e.g., A causes X causes B). These confounding factors might be the reason why some of the discovered correlations appear. A previous study by El Emam et al. (2001) found that class size can have a confounding effect on several object oriented metrics (such as some of the CK metrics suite). Specifically, they found that previously noted associations between OO metrics and fault-proneness did not exist when size is taken into account. They therefore suggest that empirical studies should control for class size when designing fault-prediction models. Such a conclusion was also shared by Zhou et al. for both fault-proneness (2014) and change-proneness (2009). Nonetheless, these conclusions are still limited to class fault- and change-proneness and the set of OO metrics used, and therefore they cannot be generalised to other factors. On the other hand, Evanco (2003) argued that introducing class size as an additional independent variable can result in models that lack internal consistency. No studies have been found to investigate the confounding effect of size in the context of unit tests and software testability.

Nevertheless, we carefully addressed this issue in more detail in Section 7.4. We applied a statistical procedure to investigate the possible indirect effect of size in regard to the discovered relationships in $RQ_{7.1}$ through $RQ_{7.5}$. Our analysis indicates that there is no statistical evidence that size is a confounding variable.

In considering the occurrence of each test and code smell it is important to highlight that these smells are measured in terms of their *presence* (or not) in a unit test or a production class. For example, a unit test can either have *Assertion Roulette* or not (*one* or *zero*), irrespective the number of times the smell was detected in the unit test. Similarly, a production class can either have (one) *Brain Class* smell or not. This approach applies to our consideration of all test or code smells. None of the test smell detection rules use size as a direct indicator of the presence of the smells (as per the test smell descriptions in Table 23). For code smells, the only smell that might be *directly* influenced by size is *Large Class*, as this smell is identified based on the size of methods within a class[45]. However, this experiment did not investigate the correlations between class size and the presence of *Large Class*, as this was not the focus of any of the research questions. For all code smells

[45] It is measured in terms of LOC of methods within a class.

(including *Large Class*), the experiment studied the correlation between the presence of the smells in a production class and the size of the associated unit tests. However, *Large Class* was not found to be correlated with any of the unit tests metrics in any of the studied systems (Table 34). Therefore, and given that there is no relationship between *Large Class* and unit test size, we conclude that size is not a confounding factor for this smell.

We further explored the relationship between all individual smells and unit test size (see $RQ_{7.6}$ and $RQ_{7.7}$) and found that size (and mainly TLOC) has *direct* effect only on *General Fixture* and *Duplicated Code* test smells (Table 33) and *Brain Class* code smell (Table 34). Other test and code smells do not appear to be directly influenced by size.

Efficacy of Smell Detection Tools

In identifying the code and test smells, the experiment depended heavily on the available smell detection tools. There is a risk that some smells were not detected by the tools used or were detected erroneously. To ensure a minimum level of accuracy tools that have been designed and examined in similar previous empirical studies were used, and for the detection of some code smells, we used an industrial quality assessment tool. For the test smell detection (and for most of the smells), an academic tool that has been successfully used in previous research was used. To maximize the accuracy of the results we verified the detected smells manually through a manual inspection process conducted by the author, and we cross-validated some of our results with those obtained for the same systems considered in prior research (Bavota et al., 2014) using those authors' publicly available data. Details of this cross-validation process are provided in Section 7.2.2.

7.6 Summary

This chapter has reported the investigation of several relationships between and among test and code smells in eight OSS that contain 1100 unit tests. The experiment looked into the possible association between the presence of smells (in unit tests and in production classes) and different software characteristics (i.e., size and complexity). It also investigated the relationship between code smells in production classes and test smells in their associated unit tests. The co-occurrence of code and test smells was also studied. In general, the results indicated a significant association between several software characteristics and test smells. We also found a significant association between the presence of code smells and class testability. In addition, some specific test smells appear to co-occur.

Chapters 5, 6 and 7 presented the results of the three experiments conducted in this thesis. The following chapter provides a collective discussion of the outcomes of the review study reported in Chapter 4 and of the three experiments presented in Chapters 5-7. It then reports the conclusions we draw from the thesis, followed by a discussion on the possible limitations of this work. We also provide a list of possible future research directions that can be followed in the light of the outcomes of this work.

Chapter 8 Conclusions and Future Work

8.1. Introduction

This thesis reports a multi-part empirical investigation of software testability and testing quality in open source OO projects. As stated in Chapter 1, the main goal of the thesis is to provide researchers and practitioners with a comprehensive understanding of design and source code factors that can affect the *testability* of a *class*. The experiments presented in this thesis have therefore been focused on unit testing, and have investigated testability not at the system level but at the class level (referred to here as class testability). Drawing on previous research in the area, class testability was measured in this work in terms of unit test size (via metrics for test lines of code (TLOC) and the number of test cases (NTC) in the unit test).

In particular, this thesis has investigated three aspects of software testability: unit test distribution and coverage, the relationship between dynamic complexity attributes of production classes and their testability, and the relationship between design flaws in both unit tests and their production classes and their class testability, through the use of the test and code smell concepts. Each of these aspects has been investigated individually in three separate experiments, presented in Chapters 5, 6 and 7, respectively.

This chapter summarises the novel elements and contributions of the thesis (Section 8.2) and presents the conclusions drawn from the results of all three experiments (Section 8.3). This is followed by a description of the limitations of the research conducted (Section 8.4) and a statement of potential future research directions (Section 8.5).

8.2. Summary of Novel Elements and Contributions

The first novel undertaking comprised a systematic mapping study on the nature and use of dynamic software metrics in software quality (Chapter 4). This study concluded that the use of dynamic metrics to measure software complexity and maintainability had been widely discussed in the literature; however the same level of attention had not been directed to the use of dynamic metrics to measure other quality attributes such as testability and reusability. An in-depth consideration of the mapping results found that complexity-related dynamic metrics, such as those said to measure coupling and cohesion, were the most widely studied, with coupling being the most often studied single metric type. Large numbers of these studies had used dynamic coupling metrics to support software comprehension and/or reengineering. The findings of this mapping study motivated the use of dynamic coupling metrics in relation to test coverage and class testability in the development and experiments that followed in Chapters 5 through 7.

A novel visualisation approach to augment specific data obtained from both static and dynamic analysis was then proposed and evaluated (Chapter 5). The long-term goal of such work is to support developers in their understanding of unit test distribution and the relationships between tests and production code. We visualised dependencies between classes using data collected based on dynamic coupling information from five OSS. The work used dependency graphs (similar to those used in the analysis of complex networks) to identify and depict dependencies between different classes in the system, and then mapped static unit test information into these dependency graphs. Besides the visual representation of the graphs, two graph *Centrality* metrics were used to further explore the distribution of unit tests within the OSS investigated.

In the experiment presented in Chapter 6 we sought to identify any relationships that might exist between two dynamic attributes of a class and its testability. The two attributes that were examined were *Dynamic Coupling* and the newly defined *Key Classes*. Dynamic metrics were used to measure these attributes due to their advantages over their static counterparts.

Finally, in Chapter 7, we studied the relationships between and among test smells, code smells and several software characteristics (such as size and complexity). The experiment resulted in a novel, in-depth investigation of factors that may impact the presence of smells in a unit test at class level, from a range of perspectives. Specifically, we investigated the relationship between different test and code smell types (individually and collectively) and the following aspects of the software code: 1) size of the unit test, 2) size and complexity of the associated production class, 3) the co-occurrence of test and code smells and 4) the co-occurrence of test smells.

8.3. Conclusions

Several conclusions are drawn from the analyses conducted and reported in Chapters 5 through 7 of this thesis.

A. Unit test distribution based on dynamic analysis

This thesis has introduced a novel visualisation approach that combines dynamic information obtained from production code with static test information to explore the distribution of unit tests in OSS. Based on the five OSS studied, it is observed that unit test and dynamic coupling information 'do not match' in that there is no significant relationship between dynamic coupling and centrality metrics and unit test coverage. In other words, unit tests do not appear to be distributed in line with the systems' dynamic coupling. Many of the central and tightly coupled classes do not come with any associated (i.e., direct) unit tests, whereas other loosely coupled classes, which are not central, appear to received direct unit testing effort.

Visualisation of the combined static unit test and dynamic coupling data provides a detailed insight into how unit tests are actually distributed in relation to the coupling level of each class in the system. The suggested visualisation and its associated *Centrality* metrics may help developers and managers to focus and optimise their test effort through the initial targeting of central system classes. Furthermore, data gathered from dynamic coupling measurement provides a comprehensive view of the dependencies of the system in relation to test information – a view that can be obtained only during software execution. Such data can be used to complement other test optimisation and prioritisation techniques to enhance future testing decisions. We believe that such visualisation techniques could be particularly helpful when developers need to maintain and reengineer existing testing suites.

B. Class testability and dynamic complexity

The resulting evidence indicates that there is a significant association between dynamic coupling and class testability. Dynamic coupling metrics, and especially the Export Coupling (EC) metric, have a significant association with Test LOC. A less significant association was found between dynamic Import Coupling (IC) and the Number of Test Cases. These results suggest that the higher the coupling between classes the larger the unit test required to test the class. Furthermore, Key Classes are shown to be significantly associated with our test suite metrics in at least three of the four systems examined. As in relation to coupling, the higher the number of executions of a class the larger the unit test required to test the class.

The findings of this experiment contribute to the general understanding of the nature of the relationships between characteristics of production and test code. The use of dynamic measures provides a level of insight that is not available using static metrics alone. These relationships should be of help in informing test maintenance and reengineering tasks.

C. Test and code smells

We conclude that the number of test smell types is associated with the size of the unit test within which they occur. We found that unit test size is associated with the presence of code and test smell types. Class testability metrics are correlated with the number of test smell types in unit tests and also with the number of code smell types in the associated production class. Also, the size and complexity of a production class are correlated with the number of test smell types in its associated unit test.

That said, the distribution of individual smell types in code is not even – we found that some smell types are more closely associated with the size of the unit test than other smell types. In particular, the occurrence of the *General Fixture* and *Duplicated Code* smell types is more strongly associated with the size of the unit test: unit tests with *General Fixture* or *Duplicated Code* are larger than unit tests without those smell types. Similarly, production classes with the *Brain Class* smell require larger unit tests than classes without this smell.

In addition to identifying several relationships between size and smells, we found that there are some test and code smell types that co-occur in production classes and their associated unit tests: the *Eager Test* smell mostly co-occurs with *Brain Class* and *Large Class* code smell types.

To summarise, the main conclusions drawn from this thesis are as follows:

- Using a combination of visualisation, dynamic analysis, static analysis and graph-based metrics it is feasible to identify central classes and to diagrammatically depict testing coverage information. Experimental results show that, even in projects with high test coverage, some classes appear to be left without direct unit testing, even though they play a central role during a typical execution profile.

- Frequently executed and tightly coupled classes are correlated with the testability of the class – such classes require larger unit tests and more test cases. This information could inform estimates of the effort required to test classes when developing new unit tests or when maintaining and refactoring existing tests.
- Test and code smells, in general, can have a negative impact on class testability. Increasing levels of size and complexity in code can have a negative impact on the presence of test smells. Classes that contain smells generally require larger unit tests, and are also likely to be associated with test smells in their associated unit tests. In addition, some particular code smells can be seen as 'signs' for the presence of test smells.

8.4. Limitations

In this thesis we conducted a series of empirical studies to address a number of questions related to software testability and testing quality.

One of the limitations of this work is that the thesis uses data obtained only from a limited set of OSS. This might limit the general applicability of the results, as industrial projects might have different characteristics compared to OSS. For example, the testing approach and methods used in industrial projects might be more rigorous and more controlled when compared with OSS. To mitigate this risk we sought to analyse as many systems as feasible within the scope of the study and to include projects that were highly mature in spite of their OSS origins.

The studies also used a limited range of test information obtained from the OSS projects. Information about the testing strategy, development method, developers' roles in their projects and developers' experience with unit testing has not been taken into consideration in this work. This is mainly because such information is not available in the documentation or source code of the projects' repositories. This also means that we had to necessarily limit our analyses to being explanatory, based on the existence and strength of association, rather than being predictive, based on causality. We have been able to rely to some extent, however, on other research that has investigated OSS testing strategies to inform the conclusions drawn.

The stated goal of the new visualisation approach developed in this thesis is to support developers in their understanding of unit test distribution. Although the approach was functionally evaluated over five different OSS, it has not been empirically assessed for utility with software professionals. As such we are not yet in a position to say whether it can deliver improved developer comprehension of unit test distribution. Such an assessment would be possible through a controlled experiment with practicing software engineers.

8.5. Future Work

Several future research endeavours could be pursued in the area of software testability based on the findings presented in this thesis.

Given the inherently limited scope of the experimental work conducted in this research it is important that those experiments be replicated using a wider range of systems (including industrial, closed-source systems), to enable further evaluation of the findings and their applicability.

It is important that the visualisation approach be tested by practicing software developers to enable us to evaluate the usefulness of the proposed approach in terms of improving program comprehension. This research could be usefully undertaken through a controlled user study with the assistance of software developers and maintainers. Future work should also investigate the *cause* of the uneven distribution of unit tests, as found in our empirical analysis.

Another research direction would be to investigate whether *Dynamic Coupling* and *Key Class* information can be used to predict the size and structure of unit tests. Predicting class testability should improve the early estimation and assessment of the effort needed in testing activities. In principle this could be achieved through the development of a regression model that uses dynamic metrics data (such as Execution Frequency and dynamic coupling metrics) to predict the effort needed to test a class (in terms of unit test design). This work could also be extended to an investigation of the association between other source code factors and testability using runtime information. It would also be potentially beneficial to incorporate information about class testability with other testing information such as test coverage and test strategy.

More generally it would be interesting to study in depth the causality of the relationships identified in this work. In terms of informing practice it would be useful to investigate which specific factors might impact the distribution of test smells in unit tests. This could be done by including information regarding the testing strategy used in a project and other indirect testing information into the mapping, to provide a more comprehensive view of testing activities and their outcomes. Consideration of a broader set of code and test smells should inform further understanding of the extent to which the results found here apply beyond the smells and systems examined. A further natural extension of the explanatory analyses reported regarding the incidence of smells would be to assess whether the relationships can inform the development of robust smell prediction models. A binary (binomial) logistic regression model could be developed to predict the presence of certain test smells (as dependent variables) based on the presence of certain code smells in the production code (i.e., using code smells as predictors/ independent variables).

In the process of conducting our work we also found many instances where developers wrote unit tests that did not include any assertions. Writing a unit test with no assertion can still give you 100% code coverage[46] but this does not reflect the ‘ground truth’ in terms of testing quality. A further specific extension of the work reported here would be to investigate the nature of Assertions in unit tests and to examine why Assertions are sometimes misused by developers in both open and closed source software.

[46] http://martinfowler.com/bliki/AssertionFreeTesting.html

References

Abbes, M., Khomh, F., Guéhéneu, Y.-G. & Antoniol, G. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. *European Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society.

Abdi, H. E. 2010. Holm's Sequential Bonferroni Procedure. *In:* Salkind, N. J. (ed.) *Encyclopedia of Research Design.* Thousand Oaks, California: SAGE Publications, Inc.

Adams, B., De Schutter, K., Zaidman, A., Demeyer, S., Tromp, H. & De Meuter, W. 2009. Using Aspect Orientation in Legacy Environments for Reverse Engineering Using Dynamic Analysis--an Industrial Experience Report. *Journal of Systems and Software,* 82**,** 668-684.

Aggarwal, K., Singh, Y. & Chhabra, J. 2003. A Dynamic Software Metric and Debugging Tool. *SIGSOFT Software Engineering Notes,* 28**,** 1.

Al Dallal, J. 2013. Object-Oriented Class Maintainability Prediction Using Internal Quality Attributes. *Information and Software Technology,* 55**,** 2028-2048.

Amalfitano, D., Fasolino, A. R., Polcaro, A. & Tramontana, P. 2010. Dynaria: A Tool for Ajax Web Application Comprehension. *International Conference on Program Comprehension (ICPC).* IEEE Computer Society.

Arisholm, E., Briand, L. C. & Foyen, A. 2004. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering,* 30**,** 491-506.

Bache, R. & Mullerburg, M. 1990. Measures of Testability as a Basis for Quality Assurance. *Software Engineering Journal,* 5**,** 86-92.

Badri, L., Badri, M. & Toure, F. 2010. Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems. *Advances in Software Engineering.* Springer.

Badri, L., Badri, M. & Toure, F. 2011. An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes. *International Journal of Software Engineering and Its Applications,* 5**,** 69-85.

Ball, T. 1999. The Concept of Dynamic Analysis. *SIGSOFT Software Engineering Notes,* 24**,** 216-234.

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**,** 751-761.

Basili, V. R. & Weiss, D. M. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering,* 10**,** 728-738.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. & Binkley, D. 2014. Are Test Smells Really Harmful? An Empirical Study. *Empirical Software Engineering***,** 1-43.

Beck, K. 1994. Simple Smalltalk Testing: With Patterns. *The Smalltalk Report,* 4**,** 16-18.

Beck, K. 2002. *Test Driven Development: By Example,* ed., Addison-Wesley Publishing Co., Inc.

Beller, M., Gousios, G., Panichella, A. & Zaidman, A. 2015. When, How, and Why Developers (Do Not) Test in Their Ides. *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE).* Bergamo, Italy.

Bertolino, A. 2007. Software Testing Research: Achievements, Challenges, Dreams. *Future of Software Engineering (FOSE).* IEEE Computer Society.

Bertolino, A. & Strigini, L. 1996. On the Use of Testability Measures for Dependability Assessment. *IEEE Transactions on Software Engineering,* 22**,** 97-108.

Biggerstaff, T. J., Mitbander, B. G. & Webster, D. 1993. The Concept Assignment Problem in Program Understanding. *International Conference on Software Engineering (ICSE).* Baltimore, Maryland, USA: IEEE Computer Society Press.

Binder, R. V. 1994. Design for Testability in Object-Oriented Systems. *Communications of the ACM,* 37**,** 87-101.

Boccaletti, S., Latora, V., Moreno, Y., Chavez, M. & Hwang, D. U. 2006. Complex Networks: Structure and Dynamics. *Physics Reports,* 424**,** 175-308.

Boehm, B., Brown, J., Kaspar, H., Lipow, M., Macleod, G. & Merrit, M. 1978. *Characteristics of Software Quality,* ed., North-Holland.

Boehm, B. W. 1981. *Software Engineering Economics,* ed., Prentice Hall PTR.

Borgatti, S. P. 2005. Centrality and Network Flow. *Social Networks,* 27**,** 55-71.

Breugelmans, M. & Van Rompaey, B. 2008. Testq: Exploring Structural and Maintenance Characteristics of Unit Test Suites. *International Workshop on Advanced Software Development Tools and Techniques.* Paphos, Cyprus.

Briand, L. C., Morasca, S. & Basili, V. R. 2002. An Operational Process for Goal-Driven Definition of Measures. *IEEE Transactions on Software Engineering,* 28**,** 1106-1125.

Brooks, F. P. 1975. *The Mythical Man-Month,* ed., Addison-Wesley Reading, MA.

Bruntink, M. & Van Deursen, A. 2006. An Empirical Study into Class Testability. *Journal of Systems and Software,* 79**,** 1219-1232.

Burrows, R., Ferrari, F. C., Garcia, A. & Taïani, F. 2010. An Empirical Evaluation of Coupling Metrics on Aspect-Oriented Programs. *International Workshop on Emerging Trends in Software Metrics (WETSoM).* Cape Town, South Africa: ACM.

Burrows, R., TaïAni, F., Garcia, A. & Ferrari, F. C. 2011. Reasoning About Faults in Aspect-Oriented Programs: A Metrics-Based Evaluation. *International Conference on Program Comprehension (ICPC).* Kingston, Ontario, Canada.

Cai, Y. 2008. Assessing the Effectiveness of Software Modularization Techniques through the Dynamics of Software Evolution. *Workshop on Assessment of COntemporary Modularization Techniques.* Orlando, US.

Cazzola, W. & Marchetto, A. 2008. Aop-Hiddenmetrics: Separation, Extensibility and Adaptability in Sw Measurement. *Journal of Object Technology,* 7**,** 53–68.

Chaumun, M. A., Kabaili, H., Keller, R. K., Lustman, F. & Saint-Denis, G. 2000. Design Properties and Object-Oriented Software Changeability. *European Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society.

Cheon, Y. & Leavens, G. 2002. A Simple and Practical Approach to Unit Testing: The Jml and Junit Way. *European Conference on Object-Oriented Programming (ECOOP).* Malaga, Spain: Springer.

Chhabra, J. K. & Gupta, V. 2010. A Survey of Dynamic Software Metrics. *Journal of Computer Science and Technology,* 25**,** 1016-1029.

Chidamber, S. R. & Kemerer, C. F. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20**,** 476-493.

Cho, E. S., Kim, C. J., Kim, S. D. & Rhew, S. Y. 1998. Static and Dynamic Metrics for Effective Object Clustering. *Asia Pacific Software Engineering Conference (APSEC).* IEEE Computer Society.

Cleland-Huang, J., Chang, C. K., Hosung, K. & Balakrishnan, A. 2001. Requirements-Based Dynamic Metrics in Object-Oriented Systems. *International Symposium on Requirements Engineering (RE).* Toronto, Canada.

Cohen, J. 1988. *Statistical Power Analysis for the Behavioral Sciences,* 2nd ed., L. Erlbaum Associates.

Coolican, H. 2014. *Research Methods and Statistics in Psychology,* 6th ed., New York, Psychology Press.

Corbi, T. A. 1989. Program Understanding: Challenge for the 1990s. *IBM Systems Journal,* 28**,** 294-306.

Cornelissen, B., Van Deursen, A., Moonen, L. & Zaidman, A. 2007. Visualizing Testsuites to Aid in Software Understanding. *European Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society.

Cornelissen, B., Zaidman, A. & Van Deursen, A. 2011. A Controlled Experiment for Program Comprehension through Trace Visualization. *IEEE Transactions on Software Engineering,* 37**,** 341-355.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L. & Koschke, R. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering,* 35**,** 684-702.

D'ambros, M., Bacchelli, A. & Lanza, M. 2010. On the Impact of Design Flaws on Software Defects. *International Conference on Quality Software (ICQS).* IEEE Computer Society.

Daniel, W. W. 2000. *Applied Nonparametric Statistics,* ed., Duxbury.

Dufour, B., Driesen, K., Hendren, L. & Verbrugge, C. 2003a. Dynamic Metrics for Java. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Anaheim, California, USA: ACM.

Dufour, B., Goard, C., Hendren, L., Moor, O. D., Sittampalam, G. & Verbrugge, C. 2004. Measuring the Dynamic Behaviour of Aspectj Programs. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Vancouver, BC, Canada: ACM.

Dufour, B., Hendren, L. & Verbrugge, C. 2003b. *J: A Tool for Dynamic Analysis of Java Programs. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Anaheim, CA, USA: ACM.

Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. 2008. Selecting Empirical Methods for Software Engineering Research. *In:* Shull, F., Singer, J. & Sjøberg, D. I. K. (eds.) *Guide to Advanced Empirical Software Engineering.* Springer London.

El Emam, K., Benlarbi, S., Goel, N. & Rai, S. N. 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering,* 27**,** 630-650.

Elberzhager, F., Rosbach, A., Münch, J. & Eschbach, R. 2012. Reducing Test Effort: A Systematic Mapping Study on Existing Approaches. *Information and Software Technology,* 54**,** 1092-1106.

Ernst, M. D. 2003. Static and Dynamic Analysis: Synergy and Duality. *Workshop on Dynamic Analysis (WODA).* Portland, US: ACM.

Evanco, W. M. 2003. Comments on "the Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". *IEEE Transactions on Software Engineering,* 29**,** 670-672.

Fenton, N. E. & Pfleeger, S. L. 1998. *Software Metrics: A Rigorous and Practical Approach,* ed., Boston, MA, USA, PWS Publishing Co.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code,* ed., Addison-Wesley Publishing Co., Inc.

Freedman, R. S. 1991. Testability of Software Components. *IEEE Transactions on Software Engineering,* 17**,** 553-564.

Freeman, L. C. 1978. Centrality in Social Networks: Conceptual Clarification. *Social Networks,* 1**,** 215-239.

Fritz, C. O., Morris, P. E. & Richler, J. J. 2012. Effect Size Estimates: Current Use, Calculations, and Interpretation. *Journal of Experimental Psychology: General,* 141**,** 2-18.

Gani, H., Ryan, C. & Rossi, P. 2006. Runtime Metrics Collection for Middleware Supported Adaptation of Mobile Applications. *Workshop on Adaptive and Reflective Middleware.* Melbourne, Australia: ACM.

Gao, J. Z., Jacob, H.-S. & Wu, Y. 2003. *Testing and Quality Assurance for Component-Based Software,* ed., Norwood, MA, USA, Artech House Publishers.

Grady, R. B. & Caswell, D. L. 1987. *Software Metrics: Establishing a Company-Wide Program,* ed., Prentice-Hall, Inc.

Graham, S. L., Kessler, P. B. & Mckusick, M. K. 1982. Gprof: A Call Graph Execution Profiler. *SIGPLAN Symposium on Compiler construction.* ACM.

Greiler, M., Van Deursen, A. & Storey, M.-A. 2013a. Automated Detection of Test Fixture Strategies and Smells. *International Conference on Software Testing, Verification and Validation (ICST).* IEEE Computer Society.

Greiler, M., Zaidman, A., Van Deursen, A. & Storey, M.-A. 2013b. Strategies for Avoiding Test Fixture Smells During Software Evolution. *Working Conference on Mining Software Repositories (MSR).* San Francisco, CA, USA: IEEE Press.

Gunnalan, R., Shereshevsky, M. & Ammar, H. H. 2005. Pseudo Dynamic Metrics [Software Metrics]. *International Conference on Computer Systems and Applications (AICCSA).* IEEE Computer Society.

Gupta, N. & Rao, P. 2001. Program Execution-Based Module Cohesion Measurement. *International Conference on Automated Software Engineering (ASE).* IEEE Computer Society.

Gupta, V. & Chhabra, J. K. 2011. Dynamic Cohesion Measures for Object-Oriented Software. *Journal of Systems Architecture,* 57**,** 452-462.

Hall, T., Zhang, M., Bowes, D. & Sun, Y. 2014. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology,* 23**,** 1-39.

Hamou-Lhadj, A. & Lethbridge, T. C. 2010. Understanding the Complexity Embedded in Large Routine Call Traces with a Focus on Program Comprehension Tasks. *IET Software,* 4**,** 161-177.

Harrison, R., Counsell, S. J. & Nithi, R. V. 1998. An Evaluation of the Mood Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering* 24**,** 491-496.

Harrold, M. J. 2000. Testing: A Roadmap. *Future of Software Engineering (FOSE).* Limerick, Ireland: ACM.

Hassoun, Y., Counsell, S. & Johnson, R. 2005. Dynamic Coupling Metric: Proof of Concept. *IEE Proceedings -Software,* 152**,** 273-279.

Hauptmann, B., Junker, M., Eder, S., Juergens, E. & Vaas, R. 2012. Can Clone Detection Support Test Comprehension? *International Conference on Program Comprehension (ICPC).*

Hayes, A. F. 2009. Beyond Baron and Kenny: Statistical Mediation Analysis in the New Millennium. *Communication Monographs,* 76**,** 408-420.

Hayes, A. F. & Preacher, K. J. 2014. Statistical Mediation Analysis with a Multicategorical Independent Variable. *British Journal of Mathematical and Statistical Psychology,* 67**,** 451-470.

IEEE. 1990. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990.

ISO. 2001. *Software Engineering - Product Quality-Part 1*. International Organization for Standardization Geneva.

Jerding, D. & Rugaber, S. 2000. Using Visualization for Architectural Localization and Extraction. *Science of Computer Programming,* 36**,** 267-284.

Jungmayr, S. 1999. Reviewing Software Artifacts for Testability. *EuroSTAR.* Barcelona, Spain.

Kan, S. H. 2002. *Metrics and Models in Software Quality Engineering,* 2nd ed., Boston, USA, Addison-Wesley.

Khomh, F., Penta, M., Guéhéneuc, Y.-G. & Antoniol, G. 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness. *Empirical Software Engineering,* 17**,** 243-275.

Khurana, P. & Kaur, P. J. 2009. Dynamic Metrics at Design Level. *International Journal of Information Technology and Knowledge Management,* 2**,** 449-454.

Kitchenham, B., Brereton, P., Turner, M., Niazi, M., Linkman, S., Pretorius, R. & Budgen, D. 2010. Refining the Systematic Literature Review Process—Two Participant-Observer Case Studies. *Empirical Software Engineering,* 15**,** 618-653.

Kitchenham, B. & Charters, S. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Keele University and University of Durham.

Kitchenham, B. A., Budgen, D. & Pearl Brereton, O. 2011. Using Mapping Studies as the Basis for Further Research – a Participant-Observer Case Study. *Information and Software Technology,* 53**,** 638-651.

Ko, A. J., Myers, B. A., Coblenz, M. J. & Aung, H. H. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering,* 32**,** 971-987.

Kochhar, P. S., Bissyande, T. F., Lo, D. & Lingxiao, J. 2013. An Empirical Study of Adoption of Software Testing in Open Source Projects. *International Conference on Quality Software (ICQS).*

Lange, D. B. & Nakamura, Y. 1997. Object-Oriented Program Tracing and Visualization. *Computer,* 30**,** 63-70.

Lanza, M. & Marinescu, R. 2006. *Object-Oriented Metrics in Practice,* ed., Springer.

Leung, H. K. N., Ligo, L. & Qu, Y. 2007. Automated Support of Software Quality Improvement. *International Journal of Quality & Reliability Management,* 24**,** 230-243.

Lo, B. W. N. & Shi, H. 1998. A Preliminary Testability Model for Object-Oriented Software. *International Conference on Software Engineering: Education & Practice.* IEEE Computer Society.

MacDonell, S., Shepperd, M., Kitchenham, B. & Mendes, E. 2010. How Reliable Are Systematic Reviews in Empirical Software Engineering? *IEEE Transactions on Software Engineering* 36**,** 676-687.

Mackinnon, D., Krull, J. & Lockwood, C. 2000. Equivalence of the Mediation, Confounding and Suppression Effect. *Prevention Science,* 1**,** 173-181.

Mackinnon, D. P., Lockwood, C. M. & Williams, J. 2004. Confidence Limits for the Indirect Effect: Distribution of the Product and Resampling Methods. *Multivariate behavioral research,* 39**,** 99-128.

Maletic, J. I., Marcus, A. & Collard, M. L. 2002. A Task Oriented View of Software Visualization. *International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT).* IEEE Computer Society.

Mäntylä, M. & Lassenius, C. 2006. Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. *Empirical Software Engineering,* 11**,** 395-431.

Mäntylä, M., Vanhanen, J. & Lassenius, C. 2003. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. *International Conference on Software Maintenance (ICSE).* IEEE Computer Society.

Marinescu, R. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *International Conference on Software Maintenance (ICSM).* IEEE Computer Society.

Mathur, R., Keen, K. J. & Etzkorn, L. H. 2010. Towards an Object-Oriented Complexity Metric at the Runtime Boundary Based on Decision Points in Code. *Annual Southeast Regional Conference.* Oxford, Mississippi: ACM.

Mayrhauser, A. V. & Vans, A. M. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer,* 28**,** 44-55.

Mccabe, T. J. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering,* SE-2**,** 308-320.

Mcmanus, J. & Wood-Harper, T. 2007. Software Engineering: A Quality Management Perspective. *The TQM Magazine,* 19**,** 315 - 327.

Mendes, E., Mosley, N. & Counsell, S. 2005. Investigating Web Size Metrics for Early Web Cost Estimation. *Journal of Systems and Software,* 77**,** 157-172.

Meszaros, G. 2006. *Xunit Test Patterns: Refactoring Test Code,* ed., Prentice Hall PTR.

Mitchell, A. & Power, J. F. 2004. Run-Time Cohesion Metrics: An Empirical Investigation. *International Conference on Software Engineering Research and Practice (SERP).* Las Vegas, USA.

Mitchell, Á. & Power, J. F. 2006. A Study of the Influence of Coverage on the Relationship between Static and Dynamic Coupling Metrics. *Science of Computer Programming,* 59**,** 4-25.

Moha, N., Gueheneuc, Y.-G., Duchien, L. & Meur, A.-F. L. 2010. Decor: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering,* 36**,** 20-36.

Moret, P., Binder, W., Heydarnoori, A. & Ansaloni, D. 2010. Tool Demonstration: Effective Runtime Exploration of the Inter-Procedural Control Flow in Java Applications. *International Conference on the Principles and Practice of Programming in Java (PPPJ).* Vienna, Austria: ACM.

Mouchawrab, S., Briand, L. C. & Labiche, Y. 2005. A Measurement Framework for Object-Oriented Software Testability. *Information and Software Technology,* 47**,** 979-997.

Munson, J. C. & Hall, G. A. 1996. Estimating Test Effectiveness with Dynamic Complexity Measurement. *Empirical Software Engineering,* 1**,** 279-305.

Munson, J. C. & Khoshgoftaar, T. M. 1992. Measuring Dynamic Program Complexity. *IEEE Software,* 9**,** 48-55.

Myers, G. J., Sandler, C. & Badgett, T. 2011. *The Art of Software Testing,* ed., Wiley Publishing.

Narasimhan, V. L. & Hendradjaya, B. 2007. Some Theoretical Considerations for a Suite of Metrics for the Integration of Software Components. *Information Sciences,* 177**,** 844-864.

Nunamaker, J. F., Chen, M. & Purdin, T. D. M. 1990. Systems Development in Information Systems Research. *Journal of Management Information Systems,* 7**,** 89-106.

Offutt, J., Abdurazik, A. & Schach, S. 2008. Quantitatively Measuring Object-Oriented Couplings. *Software Quality Journal,* 16**,** 489-512.

Osterweil, L. 1996. Strategic Directions in Software Quality. *ACM Computing Surveys,* 28**,** 738-750.

Pacione, M. J., Roper, M. & Wood, M. 2003. A Comparative Evaluation of Dynamic Visualisation Tools. *Working Conference on Reverse Engineering.* Victoria, Canada.

Patzke, T., Becker, M., Steffens, M., Sierszecki, K., Savolainen, J. E. & Fogdal, T. 2012. Identifying Improvement Potential in Evolving Product Line Infrastructures: 3 Case Studies. *International Software Product Line Conference (SPLC).* Salvador, Brazil: ACM.

Pauw, W. D., Lorenz, D., Vlissides, J. & Wegman, M. 1998. Execution Patterns in Object-Oriented Visualization. *USENIX Conference on Object-Oriented Technologies and Systems (COOTS).* Santa Fe, New Mexico: USENIX Association.

Petersen, K., Feldt, R., Mujtaba, S. & Mattsson, M. 2008. Systematic Mapping Studies in Software Engineering. *International Conference on Evaluation and Assessment in Software Engineering (EASE).* Bari, Italy.

Petersen, K., Vakkalanka, S. & Kuzniarz, L. 2015. Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update. *Information and Software Technology,* 64**,** 1-18.

Pirzadeh, H., Agarwal, A. & Hamou-Lhadj, A. 2010. An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension. *ACIS International Conference on Software Engineering Research, Management and Applications.* Montreal, Canada: IEEE Computer Society.

Preacher, K. & Hayes, A. 2004. Spss and Sas Procedures for Estimating Indirect Effects in Simple Mediation Models. *Behavior Research Methods, Instruments, & Computers,* 36**,** 717-731.

Preacher, K. & Hayes, A. 2008. Asymptotic and Resampling Strategies for Assessing and Comparing Indirect Effects in Multiple Mediator Models. *Behavior Research Methods,* 40**,** 879-891.

Putnam, L. H. 1978. A General Empirical Solution to the Macro Software Sizing and Estimating Problem. *IEEE Trans. Softw. Eng.,* 4**,** 345-361.

Qusef, A., Bavota, G., Oliveto, R., De Lucia, A. & Binkley, D. 2014. Recovering Test-to-Code Traceability Using Slicing and Textual Analysis. *Journal of Systems and Software,* 88**,** 147-168.

Razali, N. M. & Wah, Y. B. 2011. Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests. *Journal of Statistical Modeling and Analytics,* 2**,** 21-33.

Reichhart, S., Gîrba, T. & Ducasse, S. 2007. Rule-Based Assessment of Test Quality. *Journal of Object Technology,* 6**,** 231-251.

Richner, T. & Ducasse, S. 1999. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. *International Conference on Software Maintenance (ICSM).* Oxford, England, UK: IEEE Computer Society.

Riva, C. & Rodriguez, J. V. 2002. Combining Static and Dynamic Views for Architecture Reconstruction. *European Conference on Software Maintenance and Reengineering (CSMR).* Budapest, Hungary: IEEE Computer Society.

Rothlisberger, D. 2010. Exploiting Dynamic Information in Ides Eases Software Maintenance. *Workshop on Program Comprehension through Dynamic Analysis (PCODA).* Massachusetts, USA.

Rothlisberger, D., Harry, M., Villazon, A., Ansaloni, D., Binder, W., Nierstrasz, O. & Moret, P. 2009. Augmenting Static Source Views in Ides with Dynamic Metrics. *International Conference on Software Maintenance (ICSM).* Edmonton, Alberta, Canada.

Royston, P. 1992. Approximating the Shapiro-Wilk W-Test for Non-Normality. *Statistics and Computing,* 2**,** 117-119.

Runeson, P. & Andrews, A. 2003. Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. *International Symposium on Software Reliability Engineering (ISSRE).* IEEE Computer Society.

Sabane, A., Penta, M. D., Antoniol, G. & Gueheneuc, Y.-G. 2013. A Study on the Relation between Antipatterns and the Cost of Class Unit Testing. *European Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society.

Safari-Sharifabadi, E. & Constantinides, C. 2008. Dynamic Analysis of Ada Programs for Comprehension and Quality Measurement. *ACM SIGAda Ada Letters,* 28**,** 15-38.

Sarimbekov, A., Sewe, A., Kell, S., Zheng, Y., Binder, W., Bulej, L. & Ansaloni, D. 2013. A Comprehensive Toolchain for Workload Characterization across Jvm Languages. *Workshop on Program Analysis for Software Tools and Engineering (PASTE).* Seattle, Washington: ACM.

Scotto, M., Sillitti, A., Succi, G. & Vernazza, T. 2006. A Non-Invasive Approach to Product Metrics Collection. *Journal of Systems Architecture,* 52**,** 668-675.

Shtern, M., Smit, M., Simmons, B. & Litoiu, M. 2014. A Runtime Cloud Efficiency Software Quality Metric. *International Conference on Software Engineering (ICSE), New ideas and Emerging Results (NIER) track.* Hyderabad, India: ACM.

Singer, J., Lethbridge, T., Vinson, N. & Anquetil, N. 1997. An Examination of Software Engineering Work Practices. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON).* Toronto, Ontario, Canada: IBM Press.

Sjoberg, D. I., Yamashita, A., Anda, B. C. D., Mockus, A. & Dyba, T. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering,* 39**,** 1144-1156.

Solingen, R. V. & Berghout, E. 1999. *The Goal/Question/Metrics Method: A Practical Guide for Quality Improvement of Software Development,* ed., New York McGraw-Hill

Sommerville, I. 2006. *Software Engineering,* 8th ed., Boston, MA, USA, Addison-Wesley.

Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M., Mcdermid, J. & Paige, R. 2012. Large-Scale Complex It Systems. *Communications of the ACM,* 55**,** 71-77.

Stockman, S. G., Todd, A. R. & Robinson, G. A. 1990. A Framework for Software Quality Measurement. *IEEE Journal on Selected Areas in Communications,* 8**,** 224-233.

Stroulia, E. & Systä, T. 2002. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review,* 10**,** 8-17.

Systä, T., Koskimies, K. & Müller, H. 2001. Shimba—an Environment for Reverse Engineering Java Software Systems. *Software—Practice & Experience,* 31**,** 371-394.

Tahir, A., Ahmad, R. & Kasirun, Z. M. 2010. Maintainability Dynamic Metrics Data Collection Based on Aspect-Oriented Technology. *Malaysian Journal of Computer Science,* 23**,** 177-194.

Tahvildar, L. & Kontogiannis, K. 2004. Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach. *Journal of Software Maintenance and Evolution: Research and Practice,* 16**,** 331-361.

Thode, H. C. 2002. *Testing for Normality,* ed., New York, Marcel Dekker.

Traon, Y. L. & Robach, C. 1995. From Hardware to Software Testability. *International Test Conference (ITC) - Driving Down the Cost of Test.* IEEE Computer Society.

Travassos, G., Shull, F., Fredericks, M. & Basili, V. R. 1999. Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Denver, Colorado, USA: ACM.

Tsantalis, N. & Chatzigeorgiou, A. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering,* 35**,** 347-367.

Van Deursen, A., Moonen, L. M. F., Bergh, A. & Kok, G. 2001. Refactoring Test Code. *International Conference on XP and Flexible Processes in Software Engineering (XP).* Sardinia, Italy.

Van Rompaey, B., Bois, B. D., Demeyer, S. & Rieger, M. 2007. On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering,* 33**,** 800-817.

Van Rompaey, B. & Demeyer, S. 2008. Exploring the Composition of Unit Test Suites. *International Conference on Automated Software Engineering - ASE Workshops*

Van Rompaey, B. & Demeyer, S. 2009. Establishing Traceability Links between Unit Test Cases and Units under Test. *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering.* Kaiserslautern, Germany: IEEE Computer Society.

Voas, J. 1992. Dynamic Testing Complexity Metric. *Software Quality Journal,* 1**,** 101-114.

Wieringa, R., Maiden, N., Mead, N. & Rolland, C. 2006. Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion. *Requirements Engineering,* 11**,** 102-107.

Wohlin, C., Höst, M. & Henningsson, K. 2006. Empirical Research Methods in Web and Software Engineering. *In:* Mendes, E. & Mosley, N. (eds.) *Web Engineering.* Springer Berlin Heidelberg.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B. & Wessln, A. 2012. *Experimentation in Software Engineering,* ed., Springer Publishing Company, Incorporated.

Yacoub, S., Ammar, H. & Robinson, T. 1999. Dynamic Metrics for Object Oriented Designs. *International Software Metrics Symposium (METRICS).* Boca Raton, FL, USA.

Yacoub, S. M. & Ammar, H. H. 2002. A Methodology for Architecture-Level Reliability Risk Analysis. *IEEE Transactions on Software Engineering,* 28**,** 529-547.

Yamashita, A. 2014. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data. *Empirical Software Engineering,* 19**,** 1111-1143.

Yamashita, A. & Counsell, S. 2013. Code Smells as System-Level Indicators of Maintainability: An Empirical Study. *Journal of Systems and Software,* 86**,** 2639-2653.

Yamashita, A. & Moonen, L. 2013. To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? – an Empirical Study. *Information and Software Technology,* 55**,** 2223-2242.

Yuying, W., Qingshan, L., Ping, C. & Chunde, R. 2005. Dynamic Fan-in and Fan-out Metrics for Program Comprehension. *Workshop on Program Comprehension through Dynamic Analysis (PCODA).* Pennsylvania, USA.

Zaidman, A. & Demeyer, S. 2008. Automatic Identification of Key Classes in a Software System Using Webmining Techniques. *Journal of Software Maintenance and Evolution: Research and Practice* 20**,** 387-417.

Zaidman, A., Rompaey, B., Van Deursen, A. & Demeyer, S. 2011. Studying the Co-Evolution of Production and Test Code in Open Source and Industrial Developer Test Processes through Repository Mining. *Empirical Software Engineering,* 16**,** 325-364.

Zhang, M., Hall, T. & Baddoo, N. 2011. Code Bad Smells: A Review of Current Knowledge. *Journal of Software Maintenance and Evolution: Research and Practice,* 23**,** 179-202.

Zhao, L. & Elbaum, S. 2000. A Survey on Quality Related Activities in Open Source. *SIGSOFT Softw. Eng. Notes,* 25**,** 54-57.

Zhou, Y., Leung, H., Song, Q., Zhao, J., Lu, H., Chen, L. & Xu, B. 2012. An in-Depth Investigation into the Relationships between Structural Metrics and Unit Testability in Object-Oriented Systems. *Science China Information Sciences,* 55**,** 2800-2815.

Zhou, Y., Leung, H. & Xu, B. 2009. Examining the Potentially Confounding Effect of Class Size on the Associations between Object-Oriented Metrics and Change-Proneness. *IEEE Transactions on Software Engineering,* 35**,** 607-623.

Zhou, Y., Xu, B., Leung, H. & Chen, L. 2014. An in-Depth Study of the Potentially Confounding Effect of Class Size in Fault Prediction. *ACM Transactions on Software Engineering and Methodology,* 23**,** 1-51.

Appendices

Appendix A: List of Articles Found in the Pilot Study

No.	Paper	Type	Year
[M1]	(Gunnalan, et al., 2005)	Conference	2005
[M2]	(Tahir and Ahmad, 2010) [47]	Conference	2010
[M3]	(Kavitha and Shanmugam, 2008)	Conference	2008
[M4]	(Rothlisberger et al., 2009)	Conference	2009
[M5]	(Allier et al., ,2010)	Conference	2010
[M6]	(Cleland-Huang et al., ,2001)	Conference	2001
[M7]	(Zaidman et al., 2006)	Conference	2006
[M8]	(Hassoun et al., 2004)	Conference	2004
[M9]	(Gokhale and Mullen,2005)	Conference	2005
[M10]	(Hassoun et al., 2005)	Journal	2005
[M11]	(Arisholm,2002)	Conference	2002
[M12]	(Yacoub and Ammar,2002)	Journal	2002
[M13]	(Rothlisberger et al.,2009)	Conference	2009
[M14]	(Binder et al., 2007)	Conference	2007
[M15]	(Choi and Lee,2007)	Conference	2007
[M16]	(Maisikeli and Mitropoulos,2010)	Conference	2010
[M17]	(Rilling and Klemola,2003)	Conference	2003
[M18]	(Juefeng et al., 2006)	Conference	2006
[M19]	(Wu and Xu,2009)	Conference	2009
[M20]	(Alalfi et al., ,2010)	Conference	2010
[M21]	(Juan et al., 2009)	Conference	2009
[M22]	(Beszedes et al., 2007)	Conference	2007

47 Author’s previous work.

[M1] Gunnalan, R., M. Shereshevsky, and H.H. Ammar, *Pseudo dynamic metrics [software metrics]*, in *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*. 2005, IEEE Computer Society. p. 117-vii.

[M2] Tahir, A. and R. Ahmad. *An AOP-based approach for collecting software maintainability dynamic metrics*. 2010.

[M3] Kavitha, A. and A. Shanmugam. *Dynamic coupling measurement of object oriented software using trace events*. in *Applied Machine Intelligence and Informatics, 2008. SAMI 2008. 6th International Symposium on*. 2008.

[M4] Rothlisberger, D., et al. *Augmenting static source views in IDEs with dynamic metrics*. in *IEEE International Conference on Software Maintenance, 2009 (ICSM 2009)*. 2009.

[M5] Allier, S., et al., *Deriving Coupling Metrics from Call Graphs*, in *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 2010, IEEE Computer Society. p. 43-52.

[M6] Cleland-Huang, J., et al., *Requirements-Based Dynamic Metrics In Object-Oriented Systems*, in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. 2001, IEEE Computer Society. p. 212.

[M7] Zaidman, A., B.D. Bois, and S. Demeyer, *How Webmining and Coupling Metrics Improve Early Program Comprehension*, in *Proceedings of the 14th IEEE International Conference on Program Comprehension*. 2006, IEEE Computer Society. p. 74-78.

[M8] Hassoun, Y., R. Johnson, and S. Counsell, *A Dynamic Runtime Coupling Metric for Meta-Level Architectures*, in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*. 2004, IEEE Computer Society. p. 339.

[M9] Gokhale, S.S. and R.E. Mullen, *Dynamic Code Coverage Metrics: A Lognormal Perspective*, in *Proceedings of the 11th IEEE International Software Metrics Symposium*. 2005, IEEE Computer Society. p. 33.

[M10] Hassoun, Y., S. Counsell, and R. Johnson, *Dynamic coupling metric: proof of concept.* IEE Proceedings -Software, 2005. **152**(6): p. 273-279.

[M11] Arisholm, E., *Dynamic Coupling Measures for Object-Oriented Software*, in *Proceedings of the 8th International Symposium on Software Metrics*. 2002, IEEE Computer Society. p. 33.

[M12] Yacoub, S.M. and H.H. Ammar, *A Methodology for Architecture-Level Reliability Risk Analysis.* IEEE Trans. Softw. Eng., 2002. **28**(6): p. 529-547.

[M13] Rothlisberger, D., et al. *Senseo: Enriching Eclipse's static source views with dynamic metrics*. in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. 2009.

[M14] Binder, W., J. Hulaas, and P. Moret, *Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation*, in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. 2007, IEEE Computer Society. p. 91-100.

[M15] Choi, M. and J. Lee, *A Dynamic Coupling for Reusable and Efficient Software System*, in *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management \& Applications*. 2007, IEEE Computer Society. p. 720-726.

[M16] Maisikeli, S.G. and F.J. Mitropoulos. *Aspect mining using Self-Organizing Maps with method level dynamic software metrics as input vectors*. in *2nd International Conference on Software Technology and Engineering* 2010.

[M17] Rilling, J. and T. Klemola, *Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics*, in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. 2003, IEEE Computer Society. p. 115.

[M18] Juefeng, L., et al. *Improved Iterative Object-Oriented Reengineering Process based on Dynamic Coupling Measures*. in *IEEE International Conference on Systems, Man and Cybernetics*. 2006.

[M19] Wu, J. and B. Xu, *A Method to Support Web Evolution by Modeling Static Structure and Dynamic Behavior*, in *Proceedings of the 2009 International Conference on Computer Engineering and Technology - Volume 02*. 2009, IEEE Computer Society. p. 458-462.

[M20] Alalfi, M.H., J.R. Cordy, and T.R. Dean. *Automating Coverage Metrics for Dynamic Web Applications*. in *14th European Conference on Software Maintenance and Reengineering* 2010.

[M21] Juan, Z., et al. *A Dynamic Metrics Method for Test Case Reuse Based on Bayesian Network*. in *International Conference on Computational Intelligence and Software Engineering (CiSE)*. 2009.

[M22] Beszedes, A., et al., *The Dynamic Function Coupling Metric and Its Use in Software Evolution*, in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. 2007, IEEE Computer Society. p. 103-112.

Appendix B: Articles Characterisation Based on Dynamic Metrics Types

	Metrics/ Topics										
	Coupling	Cohesion	Code coverage	Complexity metrics	Code execution (i.e. unite, functions)	Memory Usage	Size/ Structure	Method/ code invocations	Polymorphism	Concurrency	Other Topics/ Metrics
[1]				√							
[2]	√	√									Modularity
[3]								√			
[4]	√			√							
[5]											Inter-processor Communication Volume (ICV)
[6]		√									
[7]	√			√							
[8]											The Most Frequently Executed Module (MFEM) Metric
[9] [10] [11]						√	√		√	√	
[12-14]	√										
[15]	√	√									
[16]	√										
[17]	√		√								
[18]						√	√				
[19]	√		√								
[20]						√	√				
[21]		√									
[22]	√										
[23]	√										
[24]	√										
[25]							√				
[26]											

	Coupling	Cohesion	Code coverage	Complexity metrics	Code execution (i.e. unite, functions)	Memory Usage	Size/ Structure	Method/ code invocations	Polymorphism	Concurrency	**Other Topics/ Metrics**
[27]	√										
[28] [29]					√			√			
[30]									√		
[31]											**Components Development:** Number of Cycle, Active Component, Average Number of Active Components
[32]	√	√	√		√	√				√	
[33]				√		√					
[34]	√										
[35]											
[36]	√	√									Modularity
[37]									√		
[38]	√										
[39]		√									
[40]	√										
[41]					√	√		√			
[42]	√										
[43]			√				√				
[44]	√	√									Method Signature, Method Spread
[45]				√							
[46]					√			√			Number of Methods Allocated
[47]					√	√		√			Number of Created Objects
[48] [48]	√										
[49]	√										
[50]	√						√				Code Churn

[48] Author's previous work.

	Coupling	Cohesion	Code coverage	Complexity metrics	Code execution (i.e. unite, functions)	Memory Usage	Size/ Structure	Method/ code invocations	Polymorphism	Concurrency	**Other Topics/ Metrics**
[51]		√									
[52]						√		√			*Method Invocation and Execution Time, Network Usage, Processor Usage*
[53]	√	√									
[54]	√				√			√			
[55]	√										
[56]											*Cloud Efficiency*
[57]		√									
[58]		√									
[59]	√	√									
[60]			√					√			
[61]											
[62]			√					√			

[1] Voas, J., Dynamic testing complexity metric. Software Quality Journal, 1992. 1(2): p. 101-114.

[2] Cho, E.S., et al., Static and dynamic metrics for effective object clustering, in Proceedings of the Fifth Asia Pacific Software Engineering Conference. 1998, IEEE Computer Society. p. 78.

[3] Richner, T. and S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, in IEEE International Conference on Software Maintenance. 1999, IEEE Computer Society: Oxford, England, UK. p. 13.

[4] Yacoub, S.M., H.H. Ammar, and T. Robinson, Dynamic metrics for object oriented designs, in International Symposium on Software Metrics. 1999, IEEE Computer Society. p. 50.

[5] Cleland-Huang, J., et al., Requirements-based dynamic metrics in object-oriented systems, in International Symposium on Requirements Engineering. 2001, IEEE Computer Society. p. 212.

[6] Gupta, N. and P. Rao. Program Execution-Based Module Cohesion Measurement. in International Conference on Automated Software Engineering. 2001. IEEE Computer Society.

[7] Yacoub, S.M. and H.H. Ammar, A Methodology for Architecture-Level Reliability Risk Analysis. IEEE Transaction on Software Engineering, 2002. 28(6): p. 529-547.

[8] Aggarwal, K.K., Y. Singh, and J.K. Chhabra, A dynamic software metric and debugging tool. SIGSOFT Software Engineering Notes, 2003. 28(2): p. 1.

[9] Dufour, B., et al., Dynamic metrics for java, in ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. 2003, ACM: Anaheim, California, USA. p. 149-168.

[10] Dufour, B., L. Hendren, and C. Verbrugge, *J: a tool for dynamic analysis of Java programs, in Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2003, ACM: Anaheim, CA, USA. p. 306-307.

[11] Dufour, B., L. Hendren, and C. Verbrugge, Problems in Objectively Quantifying Benchmarks using Dynamic Metrics. 2003, Sable Research Group, School of Computer Science ,McGill University. p. 8.

[12] Mitchell, A. and J.F. Power, Run-time Coupling Metrics for the Analysis of Java Programs - preliminary results from the SPEC and Grande suites. 2003, Department of Computer Science, National University of Ireland.

[13] Mitchell, A. and J.F. Power, Using object-level run-time metrics to study coupling between objects, in ACM Symposium on Applied Computing. 2005, ACM: Santa Fe, New Mexico. p. 1456-1462.

[14] Mitchell, Á. and J.F. Power, An empirical investigation into the dimensions of run-time coupling in Java programs, in International Symposium on Principles and Practice of Programming in Java. 2004, Trinity College Dublin: Las Vegas, Nevada. p. 9-14.

[15] Mitchell, A. and J.F. Power, Toward a definition of run-time object-oriented metrics, in Workshop on Quantitative Approaches in Object-Oriented Software Engineering. 2003: Darmstadt, Germany.

[16] Arisholm, E., L.C. Briand, and A. Foyen, Dynamic Coupling Measurement for Object-Oriented Software. IEEE Transactions on Software Engineering, 2004. 30(8): p. 491-506.

[17] Dufour, B., et al., Measuring the dynamic behaviour of AspectJ programs, in Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2004, ACM: Vancouver, BC, Canada. p. 150-169.

[18] Mitchell, A. and J.F. Power, An approach to quantifying the run-time behaviour of Java GUI applications, in Winter International Synposium on Information and Communication Technologies. 2004, Trinity College Dublin: Cancun, Mexico. p. 1-6.

[19] Mitchell, Á. and J.F. Power, A study of the influence of coverage on the relationship between static and dynamic coupling metrics. Science of Computer Programming, 2006. 59(1-2): p. 4-25.

[20] Mitchell, A. and J.F. Power, An approach to quantifying the run-time behaviour of Java GUI applications, in Proceedings of the winter international synposium on Information and communication technologies. 2004, Trinity College Dublin: Cancun, Mexico. p. 1-6.

[21] Mitchell, A. and J.F. Power, Run-Time Cohesion Metrics: An Empirical Investigation, in International Conference on Software Engineering Research and Practice. 2004: Las Vegas, USA.

[22] Zaidman, A. and S. Demeyer, Analyzing large event traces with the help of coupling metrics, in International Workshop on Object-Oriented Reengineering. 2004: Antwerp, Belgium.

[23] Gunnalan, R., M. Shereshevsky, and H.H. Ammar, Pseudo dynamic metrics [software metrics], in International Conference on Computer Systems and Applications. 2005, IEEE Computer Society. p. 117-vii.

[24] Hassoun, Y., S. Counsell, and R. Johnson, Dynamic coupling metric: proof of concept. IEE Proceedings -Software, 2005. 152(6): p. 273-279.

[25] Mendes, E., N. Mosley, and S. Counsell, Investigating Web size metrics for early Web cost estimation. Journal Systems and Software, 2005. 77(2): p. 157-172.

[26] Yuying, W., et al., Dynamic Fan-in and Fan-out Metrics for Program Comprehension, in International Workshop on Program Comprehension through Dynamic Analysis. 2005: Pittsburgh, Pennsylvania, USA.

[27] Beszedes, A., et al., The Dynamic Function Coupling Metric and Its Use in Software Evolution, in European Conference on Software Maintenance and Reengineering. 2007, IEEE Computer Society. p. 103-112.

[28] Binder, W., J. Hulaas, and P. Moret, Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation, in International Working Conference on Source Code Analysis and Manipulation. 2007, IEEE Computer Society. p. 91-100.

[29] Binder, W., et al. Towards a domain-specific aspect language for dynamic program analysis. in Annual Workshop on Domain-Specific Aspect Languages. 2011.

[30] Choi, K.H.T. and E. Tempero, Dynamic measurement of polymorphism, in Australasian Conference on Computer Science. 2007, Australian Computer Society, Inc.: Ballarat, Victoria, Australia. p. 211-220.

[31] Narasimhan, V.L. and B. Hendradjaya, Some theoretical considerations for a suite of metrics for the integration of software components. Information Sciences, 2007. 177(3): p. 844-864.

[32] Cazzola, W. and A. Marchetto, AOP-HiddenMetrics: Separation, Extensibility and Adaptability in SW Measurement. Journal of Object Technology, 2008. 7(2): p. 53–68.

[33] Keen, K.J., R. Mathur, and L. Etzkorn, Towards a measure of software intelligence employing a runtime complexity metric, in International Conference on Software Engineering and Applications. 2009: Cambridge, MA, USA.

[34] Zaidman, A. and S. Demeyer, Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice 2008. 20(6): p. 387-417.

[35] Gupta, V. and J.K. Chhabra, Measurement of dynamic metrics using dynamic analysis of programs, in WSEAS International Conference on Applied Computing Conference. 2008, World Scientific and Engineering Academy and Society (WSEAS): Istanbul, Turkey. p. 81-86.

[36] Safari-Sharifabadi, E. and C. Constantinides, Dynamic analysis of Ada programs for comprehension and quality measurement, in SIGAda Annual International Conference. 2008, ACM: Portland, OR, USA. p. 15-38.

[37] Sandhu, P.S. and G. Singh, Dynamic Metrics for Polymorphism in Object Oriented Systems. World Academy of Science, Engineering and Technology, 2008. 39(67).

[38] Adams, B., et al., Using aspect orientation in legacy environments for reverse engineering using dynamic analysis-An industrial experience report. Journal of Systems and Software, 2009. 82(4): p. 668-684.

[39] Khurana, P. and P.J. Kaur, Dynamic Metrics at design Level. International Journal of Information Technology and Knowledge Management, 2009. 2(2): p. 449-454.

[40] Quynh, P.T. and H.Q. Thang, Dynamic Coupling Metrics for Service-Oriented Software. International Journal of Computer Science and Engineering, 2009. 3(1): p. 6.

[41] Rothlisberger, D., et al. Augmenting static source views in IDEs with dynamic metrics. in International Conference on Software Maintenance. 2009.

[42] Allier, S., et al., Deriving Coupling Metrics from Call Graphs, in International Working Conference on Source Code Analysis and Manipulation. 2010, IEEE Computer Society. p. 43-52.

[43] Amalfitano, D., et al., DynaRIA: A Tool for Ajax Web Application Comprehension, in International Conference on Program Comprehension. 2010, IEEE Computer Society. p. 46-47.

[44] Maisikeli, S.G. and F.J. Mitropoulos. Aspect mining using Self-Organizing Maps with method level dynamic software metrics as input vectors. in International Conference on Software Technology and Engineering. 2010.

[45] Mathur, R., K.J. Keen, and L.H. Etzkorn, Towards an object-oriented complexity metric at the runtime boundary based on decision points in code, in Annual Southeast Regional Conference. 2010, ACM: Oxford, Mississippi. p. 1-5.

[46] Moret, P., et al., Tool demonstration: effective runtime exploration of the inter-procedural control flow in Java applications, in International Conference on the Principles and Practice of Programming in Java. 2010, ACM: Vienna, Austria. p. 162-165.

[47] Röthlisberger, D., Exploiting Dynamic Information in IDEs Eases Software Maintenance, in International Workshop on Program Comprehension through Dynamic Analysis. 2010: Massachusetts, USA.

[48] Tahir, A., R. Ahmad, and Z.M. Kasirun, Maintainability dynamic metrics data collection based on aspect-oriented technology. Malaysian Journal of Computer Science, 2010. 23(3): p. 177-194.

[49] Babu, S. and R.M.S. Parvathi, Design dynamic coupling measurement of distributed object oriented software using trace events. Journal of Computer Science, 2011. 7(5): p. 770-778.

[50] Burrows, R., et al. Reasoning about Faults in Aspect-Oriented Programs: A Metrics-Based Evaluation. in International Conference on Program Comprehension. 2011.

[51] Gupta, V. and J.K. Chhabra, Dynamic cohesion measures for object-oriented software. Journal of Systems Architecture, 2011. 57(4): p. 452-462.

[52] Gani, H., C. Ryan, and P. Rossi, Runtime metrics collection for middleware supported adaptation of mobile applications, in Workshop on Adaptive and Reflective Middleware. 2006, ACM: Melbourne, Australia. p. 2.

[53] Tosi, D. and A. Tahir, A Survey on How Well-Known Open Source Software Projects Are Tested, in Software and Data Technologies, J. Cordeiro, M. Virvou, and B. Shishkov, Editors. 2013, Springer Berlin Heidelberg. p. 42-57.

[54] Tahir, A., S.G. MacDonell, and J. Buchan, Understanding class-level testability through dynamic analysis, in International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). 2014: Lisbon, Portugal. p. 38-47.

[55] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, An empirical study on the developers’ perception of software coupling, in Proceedings of the 2013 International Conference on Software Engineering. 2013, IEEE Press: San Francisco, CA, USA. p. 692-701.

[56] Shtern, M., et al., A runtime cloud efficiency software quality metric, in Companion Proceedings of the 36th International Conference on Software Engineering. 2014, ACM: Hyderabad, India. p. 416-419.

[57] Andras, P., et al. A measure to assess the behavior of method stereotypes in object-oriented software. in 2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM). 2013.

[58] Mathur, R., K.J. Keen, and L.H. Etzkorn, Towards a measure of object oriented runtime cohesion based on number of instance variable accesses, in Proceedings of the 49th Annual Southeast Regional Conference. 2011, ACM: Kennesaw, Georgia. p. 255-257.

[59] Dugerdil, P. and M. Niculescu. Visualizing Software Structure Understandability. in Software Engineering Conference (ASWEC), 2014 23rd Australian. 2014.

[60] Sarimbekov, A., et al., A comprehensive toolchain for workload characterization across JVM languages, in Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2013, ACM: Seattle, Washington. p. 9-16.

[61] Lavazza, L., et al., On the definition of dynamic software measures, in Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement. 2012, ACM: Lund, Sweden. p. 39-48.

[62] Sarimbekov, A., et al., Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. Electronic Notes in Theoretical Computer Science, 2011. 279(1): p. 61-74.

Appendix C: Centrality Metrics Values

Appendix Table I: Centrality metrics values for FindBugs

DC: Degree Centrality, **BC**: Betweeness Centrality

Class	DC	BC	Unit test	Class	DC	BC	Unit test
AbstractBugReporter	11	1697	No	detect.LazyInit	29	732	No
analysis.AnnotatedObject	1	0	No	detect.LoadOfKnownNullValue	10	205	No
analysis.AnnotationValue	8	31	No	detect.LostLoggerDueToWeakReference	3	2	No
analysis.ClassData	4	54	No	detect.MethodReturnCheck	10	182	No
analysis.ClassInfo	16	389	No	detect.MethodReturnValueStreamFactory	5	18	No
analysis.ClassNameAndSuperclassInfo	1	0	No	detect.Methods	1	0	No
analysis.EnumValue	1	0	No	detect.MultithreadedInstanceAccess	1	0	No
analysis.FieldInfo	10	92	No	detect.MutableStaticFields	10	83	No
analysis.MethodInfo	40	4531	No	detect.Naming	12	112	No
AnalysisCacheToRepositoryAdapter	4	10	No	detect.NoteAnnotationRetention	4	105	No
AnalysisLocal	5	75	No	detect.NoteCheckReturnValue Annotations	2	0	No
Analyze	9	790	No	detect.NoteDirectlyRelevantTypeQualifiers	6	676	No
AppVersion	2	2	No	detect.NoteJCIPAnnotation	2	0	No
asm.ClassNodeAnalysisEngine	1	0	No	detect.NoteNonnullReturnValues	1	0	No
asm.ClassNodeDetector	3	163	No	detect.NoteSuppressedWarnings	3	5	No
asm.ClassReaderAnalysisEngine	2	7	No	detect.NoteUnconditionalParamDerefs	1	0	No
asm.EngineRegistrar	1	0	No	detect.NumberConstructor	7	18	No
asm.FBClassReader	2	0	No	detect.OverridingEqualsNotSymmetrical	9	341	No
ba.AbstractBlockOrder	1	0	No	detect.PreferZeroLengthArrays	3	5	No
ba.AbstractClassMember	1	0	No	detect.QuestionableBooleanAssignment	2	3	No
ba.AbstractDataflow	1	0	No	detect.ReadOfInstanceFieldInMethodInvokedByConstructorInSuperclass	14	833	No
ba.AbstractDataflowAnalysis	3	20	No	detect.ReadReturnShouldBeChecked	4	14	No
ba.AbstractDominatorsAnalysis	3	638	No	detect.RedundantInterfaces	3	7	No
ba.AbstractFrameModelingVisitor	1	0	No	detect.ReflectiveClasses	4	4	No
ba.AbstractMethod	4	11	No	detect.RepeatedConditionals	5	15	No
ba.AnalysisContext	105	18297	No	detect.RuntimeExceptionCapture	18	196	No
ba.AnnotationDatabase	8	111	No	detect.SerializableIdiom	18	575	No
ba.AnnotationRetentionDatabase	2	2	No	detect.StartInConstructor	5	19	No
ba.AssertionMethods	4	1	No	detect.StaticCalendarDetector	14	771	No
ba.BasicAbstractDataflowAnalysis	4	640	No	detect.StaticFieldLoadStreamFactory	2	0	No
ba.BasicBlock	28	1534	No	detect.Stream	13	443	No
ba.BetterCFGBuilder2	11	1603	No	detect.StreamEquivalenceClass	3	0	No
ba.BlockOrder	1	0	No	detect.StreamEscape	2	0	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
ba.BytecodeScanner	3	8	No	detect.StreamFactory	1	0	No
ba.CFG	51	6137	No	detect.StreamFrameModelingVisitor	5	63	No
ba.CFGBuilder	1	0	No	detect.StreamResourceTracker	6	648	No
ba.CFGBuilderFactory	1	0	No	detect.StringConcatenation	4	33	No
ba.ch.Subtypes2	3	15	No	detect.SuperfluousInstanceOf	3	92	No
ba.CheckReturnAnnotationDatabase	3	25	No	detect.SwitchFallthrough	10	155	No
ba.ClassContext	86	17231	No	detect.SynchronizationOnSharedBuiltinConstant	7	20	No
ba.ClassMember	1	0	No	detect.SynchronizeAndNullCheckField	1	0	No
ba.ClassNotFoundExceptionParser	5	14	No	detect.UncallableMethodOfAnonymousClass	10	95	No
ba.ClassSummary	4	2	No	detect.UnreadFields	30	299	No
ba.Dataflow	15	2685	No	detect.UnreadFieldsData	6	3	No
ba.DataflowAnalysis	1	0	No	detect.URLProblems	2	1	No
ba.DataflowValueChooser	1	0	No	detect.VarArgsProblems	2	3	No
ba.Debug	1	0	No	detect.VolatileUsage	7	44	No
ba.DefaultNullnessAnnotations	4	19	No	detect.WaitInLoop	2	3	No
ba.DepthFirstSearch	4	639	No	detect.WrongMapIterator	7	40	No
ba.DominatorsAnalysis	2	5	No	detect.XMLFactoryBypass	2	4	No
ba.Edge	20	1297	No	Detector	1	0	No
ba.EdgeChooser	1	0	No	Detector2	2	22	No
ba.EqualsKindSummary	2	0	No	DetectorFactory	15	230	No
ba.ExceptionHandlerMap	2	0	No	DetectorFactoryChooser	1	0	No
ba.FieldSummary	20	814	No	DetectorFactoryCollection	25	3770	Yes
ba.Frame	6	1440	No	DetectorToDetector2Adapter	4	641	No
ba.FrameDataflowAnalysis	4	61	No	engine.asm.EngineRegistrar	1	0	No
ba.Hierarchy	42	4998	No	engine.bcel.EngineRegistrar	1	0	No
ba.Hierarchy2	27	1321	No	engine.ClassDataAnalysisEngine	6	339	No
ba.IncompatibleTypes	14	105	No	engine.ClassInfoAnalysisEngine	5	202	No
ba.InnerClassAccess	4	31	No	engine.ClassNameAndSuperclassInfoAnalysisEngine	1	0	No
ba.InnerClassAccessMap	5	20	No	engine.ClassParser	3	0	No
ba.INullnessAnnotationDatabase	9	39	No	engine.ClassParserInterface	2	8	No
ba.JavaClassAndMethod	10	251	No	engine.ClassParserUsingASM	3	0	No
ba.JavaClassAndMethodChooser	2	0	No	engine.EngineRegistrar	2	404	No
ba.JCIPAnnotationDatabase	2	3	No	engine.SelfMethodCalls	5	477	No
ba.LiveLocalStoreAnalysis	3	1	No	EqualsOperandShouldHaveClassCompatibleWithThis	10	122	No
ba.LiveLocalStoreDataflow	2	638	No	ErrorCountingBugReporter	2	15	No
ba.Location	48	5088	No	FieldAnnotation	21	1128	No
ba.LockAnalysis	6	55	No	filter.AndMatcher	2	6	No
ba.LockChecker	10	757	No	filter.BugMatcher	10	1490	No
ba.LockDataflow	6	648	No	filter.CompoundMatcher	3	100	No
ba.LockSet	11	112	No	filter.Filter	6	244	No
ba.MethodBytecodeSet	1	0	No	filter.Matcher	6	192	No
ba.MethodUnprofitableException	4	20	No	filter.StringSetMatch	1	0	No
ba.MissingClassException	1	0	No	FilterBugReporter	3	194	No
ba.NullnessAnnotationDatabase	3	1	No	FindBugs	17	350	No
ba.ObjectTypeFactory	13	494	No	FindBugs2	44	141	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
ba.OpcodeStackScanner	3	6	No	FindBugsAnalysisFeatures	7	117	No
ba.Path	6	772	No	FindBugsDisplayFeatures	3	61	No
ba.PathVisitor	1	0	No	FindBugsMessageFormat	7	63	No
ba.PostDominatorsAnalysis	2	5	No	FindBugsProgress	1	0	No
ba.PruneInfeasibleExceptionEdges	6	88	No	Footprint	1	0	No
ba.PruneUnconditionalExceptionThrowerEdges	13	211	No	formatStringChecker.Formatter	1	0	No
ba.PutfieldScanner	1	0	No	generic.GenericObjectType	8	28	Yes
ba.RepositoryLookupFailureCallback	9	176	No	generic.GenericSignatureParser	4	3	Yes
ba.ResourceTracker	2	1	No	generic.GenericUtilities	8	46	Yes
ba.ResourceValue	4	10	No	graph.AbstractDepthFirstSearch	3	956	No
ba.ResourceValueAnalysis	10	1594	No	graph.AbstractEdge	2	0	No
ba.ResourceValueFrame	6	84	No	graph.AbstractGraph	2	0	No
ba.ResourceValueFrameModelingVisitor	3	1	No	graph.AbstractVertex	2	0	No
ba.ReverseDepthFirstSearch	2	638	No	graph.DepthFirstSearch	2	317	No
ba.SignatureConverter	9	96	No	graph.Graph	2	1	No
ba.SignatureParser	15	300	Yes	graph.GraphEdge	3	254	No
ba.SourceFile	1	0	No	graph.GraphVertex	1	0	No
ba.SourceFinder	5	107	No	graph.ReverseDepthFirstSearch	2	318	No
ba.SourceInfoMap	4	27	No	gui.AnnotatedString	3	140	No
ba.Target	1	0	No	gui2.AboutDialog	2	0	No
ba.TargetEnumeratingVisitor	1	0	No	gui2.AnalyzingDialog	6	783	No
ba.TestCaseDetector	6	4	No	gui2.BugAspects	6	143	No
ba.UnresolvedXField	4	9	No	gui2.BugLeafNode	10	406	No
ba.UnresolvedXMethod	4	9	No	gui2.BugLoader	11	577	No
ba.URLClassPath	1	0	No	gui2.BugRenderer	2	0	No
ba.XClas	1	0	No	gui2.BugSaver	2	9	No
ba.XClass	41	1591	No	gui2.BugSet	15	1033	No
ba.Xclass	2	1	No	gui2.BugTreeModel	12	785	No
ba.XFactory	67	7563	No	gui2.CheckBoxList	1	0	No
ba.Xfactory	4	43	No	gui2.CloudCommentsPane	10	130	No
ba.XField	28	847	No	gui2.CloudCommentsPaneSwing	1	0	No
ba.XMethod	57	3663	No	gui2.CommentsArea	5	643	No
ba.Xmethod	3	4	No	gui2.Debug	5	33	No
ba.XMethodParameter	2	1	No	gui2.Driver	20	4547	No
bcel.AnalysisFactory	1	0	No	gui2.FBDialog	1	0	No
bcel.AssertionMethodsFactory	1	0	No	gui2.FBFileChooser	4	120	No
bcel.BCELUtil	11	184	No	gui2.FBFrame	1	0	No
bcel.bcelUtil	1	0	No	gui2.FilterActivity	4	640	No
bcel.CFGDetector	4	6	No	gui2.FilterFactory	4	25	Yes
bcel.CFGFactory	15	2121	No	gui2.FilterFromBugPicker	3	8	No
bcel.ClassContextClassAnalysisEngine	2	0	No	gui2.FilterMatcher	3	638	No
bcel.ConstantDataflowFactory	1	0	No	gui2.FindBugsFileFilter	2	0	No
bcel.ConstantPoolGenFactory	1	0	No	gui2.FindBugsLayoutManager	3	1	No
bcel.DepthFirstSearchFactory	1	0	No	gui2.FindBugsLayoutManagerFactory	1	0	No
bcel.DominatorsAnalysisFactory	1	0	No	gui2.GUI2CommandLine	1	0	No
bcel.EngineRegistrar	2	2	No	gui2.GUISaveState	15	168	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
bcel.IsNullValueDataflowFactory	3	6	No	gui2.GuiUtil	1	0	No
bcel.JavaClassAnalysisEngine	4	92	No	gui2.MainFrame	39	888	No
bcel.LiveLocalStoreDataflowFactory	1	0	No	gui2.MainFrameComponentFactory	15	125	No
bcel.LoadedFieldSetFactory	4	127	No	gui2.MainFrameHelper	5	11	No
bcel.LockCheckerFactory	1	0	No	gui2.MainFrameLoadSaveHelper	7	483	No
bcel.LockDataflowFactory	1	0	No	gui2.MainFrameMenu	16	372	No
bcel.MethodFactory	3	13	No	gui2.MainFrameTree	12	197	No
bcel.MethodGenFactory	4	14	No	gui2.NewFilterFromBug	4	719	No
bcel.NonExceptionPostdominatorsAnalysisFactory	1	0	No	gui2.NewProjectWizard	9	277	No
bcel.ObligationDataflowFactory	7	697	No	gui2.PluginUpdateDialog	3	18	No
bcel.OpcodeStackDetector	2	0	No	gui2.PreferencesFrame	13	774	No
bcel.PreorderDetector	1	0	No	gui2.ProjectSettings	2	0	No
bcel.ReturnPathTypeDataflowFactory	1	0	No	gui2.RecentMenu	5	67	No
bcel.ReverseDepthFirstSearchFactory	1	0	No	gui2.SaveType	4	300	Yes
bcel.TypeDataflowFactory	3	2	No	gui2.Sortables	14	290	No
bcel.UnconditionalValueDerefDataflowFactory	3	19	No	gui2.SortableStringComparator	2	0	No
bcel.UnpackedBytecodeCallback	2	638	No	gui2.SorterDialog	4	7	No
bcel.UnpackedCode	1	0	No	gui2.SorterTableColumnModel	6	7	No
bcel.UsagesRequiringNonNullValuesFactory	2	4	No	gui2.SourceCodeDisplay	10	271	No
bcel.ValueNumberDataflowFactory	3	6	No	gui2.SplashFrame	1	0	No
bcp.Binding	3	0	No	gui2.SplitLayout	4	1	No
bcp.BindingSet	3	0	No	gui2.StackedFilterMatcher	1	0	No
bcp.ByteCodePattern	3	7	No	gui2.ViewFilter	9	497	No
bcp.ByteCodePatternMatch	2	0	No	gui2.WideComboBox	1	0	No
bcp.FieldAccess	4	14	No	I18N	13	106	No
bcp.FieldVariable	2	9	No	IClassScreener	1	0	No
bcp.IfNull	3	12	No	IFindBugsEngine	2	2	No
bcp.Load	3	16	No	IGuiCallback	1	0	No
bcp.LocalVariable	1	0	No	impl.AbstractScannableCodeBase	2	638	No
bcp.OneVariableInstruction	1	0	No	impl.AbstractScannableCodeBaseEntry	1	0	No
bcp.PatternElement	9	1810	No	impl.AnalysisCache	10	348	No
bcp.PatternElementMatch	3	2	No	impl.ClassFactory	2	638	No
bcp.PatternMatcher	8	462	No	impl.ClassPathBuilder	17	554	No
bcp.Store	3	16	No	impl.ClassPathImpl	1	0	No
bcp.Variable	1	0	No	impl.FilesystemCodeBaseLocato	1	0	No
bcp.Wild	1	0	No	impl.FilesystemCodeBaseLocator	1	0	No
BugAccumulator	52	3247	No	impl.ZipCodeBaseFactory	4	128	No
BugAnnotation	6	24	No	impl.ZipFileCodeBaseEntry	1	0	No
BugAnnotationUtil	7	45	No	indbugs.DetectorFactoryCollection	1	0	No
BugAnnotationWithSourceLines	4	13	No	IntAnnotation	5	17	Yes
BugCategory	3	5	No	interproc.MethodPropertyDatabase	3	0	No
BugCode	2	0	No	interproc.ParameterProperty	5	2	Yes

Class	DC	BC	Unit test	Class	DC	BC	Unit test
BugCollection	14	1026	No	interproc.PropertyDatabase	2	9	No
BugCollectionBugReporter	4	646	No	io.IO	4	112	Yes
BugInstance	131	33803	No	JavaVersion	3	18	No
BugPattern	7	69	No	jsr305.AbstractMethodAnnotationAccumulator	1	0	No
BugProperty	3	1	Yes	jsr305.Analysis	5	75	No
BugRanke	1	0	No	jsr305.DirectlyRelevantTypeQualifiersDatabase	1	0	No
BugRanker	17	1990	No	jsr305.FindBugsDefaultAnnotations	1	0	No
BugReporter	62	6727	No	jsr305.JSR305NullnessAnnotations	1	0	No
BugReporterObserver	1	0	No	jsr305.ParameterAnnotationAccumulator	1	0	No
ByteCodePatternDetector	3	1275	No	jsr305.ParameterAnnotationLookupResult	1	0	No
BytecodeScanningDetector	4	23	No	jsr305.ReturnTypeAnnotationAccumulator	1	0	No
CallGraph	4	1	No	jsr305.ReturnTypeAnnotationLookupResult	2	0	No
CallGraphEdge	2	0	No	jsr305.TypeQualifierAnnotation	7	707	No
CallGraphNode	3	0	No	jsr305.TypeQualifierAnnotationLookupResult	1	0	No
CallSite	3	1	No	jsr305.TypeQualifierApplications	29	382	No
ch.ClassVertex	3	11	No	jsr305.TypeQualifierResolver	8	34	No
ch.InheritanceEdge	1	0	No	jsr305.TypeQualifierValue	17	594	No
ch.InheritanceGraph	2	0	No	L10N	16	276	No
ch.InheritanceGraphVisitor	1	0	No	LaunchAppropriateUI	1	0	No
ch.OverriddenMethodsVisitor	3	1	No	LocalVariableAnnotation	23	144	No
ch.Subtypes2	46	4154	No	log.Profiler	15	139	No
charsets.UTF8	5	775	No	log.YourKitController	1	0	No
CheckBcel	1	0	No	Lookup	7	91	No
ClassAnnotation	14	375	No	MethodAnnotation	28	120	No
classfile.ClassDescriptor	55	5983	No	npe.DerefFinder	21	838	No
classfile.DescriptorFactory	37	3392	No	npe.IsNullConditionDecision	3	10	No
classfile.FieldDescriptor	6	23	No	npe.IsNullValue	16	238	Yes
classfile.FieldOrMethodDescriptor	4	1	No	npe.IsNullValueAnalysis	18	827	No
classfile.Global	57	5487	No	npe.IsNullValueAnalysisFeatures	1	0	No
classfile.IAnalysisCache	70	11231	No	npe.IsNullValueDataflow	11	126	No
classfile.IAnalysisEngine	1	0	No	npe.IsNullValueFrame	14	177	No
classfile.IClassAnalysisEngine	4	1023	No	npe.IsNullValueFrameModelingVisitor	17	546	No
classfile.IClassFactory	2	24	No	npe.LocationWhereValueBecomesNull	3	4	No
classfile.IClassObserver	1	0	No	npe.NullDerefAndRedundantComparisonCollector	1	0	No
classfile.IClassPath	3	45	No	npe.NullDerefAndRedundantComparisonFinder	27	317	No
classfile.IClassPathBuilder	1	0	No	npe.NullValueUnconditionalDeref	3	0	No
classfile.IClassPathBuilderProgress	1	0	No	npe.ParameterNullnessPropertyDatabase	5	12	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
classfile.ICodeBase	3	662	No	npe.PointerUsageRequiringNonNullValue	2	0	No
classfile.ICodeBaseEntry	2	4	No	npe.RedundantBranch	1	0	No
classfile.ICodeBaseIterator	1	0	No	npe.ReturnPathType	2	25	Yes
classfile.ICodeBaseLocator	1	0	No	npe.ReturnPathTypeAnalysis	3	20	No
classfile.IDatabaseFactory	1	0	No	npe.ReturnPathTypeDataflow	2	638	No
classfile.IErrorLogger	3	10	No	npe.ReturnValueNullnessPropertyDatabase	2	0	No
classfile.IMethodAnalysisEngine	2	260	No	npe.TypeQualifierNullnessAnnotationDatabase	16	469	No
classfile.impl.ClassFactory	1	0	No	npe.UsagesRequiringNonNullValues	8	58	No
classfile.IScannableCodeBase	1	0	No	obl.InstructionActionCache	14	127	No
classfile.MethodDescriptor	18	1778	No	obl.MatchMethodEntry	2	1	No
classfile.MissingClassException	5	43	No	obl.Obligation	4	13	No
classfile.ReflectionDatabaseFactory	1	0	No	obl.ObligationAnalysis	19	313	No
classfile.ResourceNotFoundException	3	0	No	obl.ObligationDataflow	1	0	No
cloud.Cloud	8	92	No	obl.ObligationFactory	15	912	No
cloud.CloudFactory	7	158	No	obl.ObligationPolicyDatabase	8	665	No
cloud.CloudPlugin	2	2	No	obl.ObligationPolicyDatabaseAction	4	645	No
cloud.CloudPluginBuilder	2	8	No	obl.ObligationPolicyDatabaseActionType	1	0	No
cloud.CloudPlugincloud.CloudPlugi	1	0	No	obl.ObligationPolicyDatabaseEntry	1	0	No
cloud.DoNothingCloud	5	330	No	obl.ObligationSet	5	40	No
ComponentPlugin	2	0	No	obl.State	4	2	No
config.AnalysisFeatureSetting	2	0	No	obl.StateSet	5	8	No
config.ProjectFilterSettings	1	0	Yes	OpcodeStack	55	825	Yes
config.UserPreferences	6	652	Yes	PackageMemberAnnotation	8	219	No
constant.Constant	4	294	No	PackageStats	4	0	No
constant.ConstantAnalysis	4	166	No	plan.AnalysisPass	3	8	No
constant.ConstantDataflow	3	638	No	plan.ConstraintEdge	1	0	No
constant.ConstantFrame	4	294	No	plan.ConstraintGraph	1	0	No
constant.ConstantFrameModelingVisitor	3	0	No	plan.DetectorFactorySelector	1	0	No
DeepSubtypeAnalysis	10	907	No	plan.DetectorNode	1	0	No
DelegatingBugReporter	1	0	No	plan.DetectorOrderingConstraint	2	22	No
deref.UnconditionalValueDerefAnalysis	33	2852	No	plan.ExecutionPlan	12	728	No
deref.UnconditionalValueDerefDataflow	4	20	No	plan.ReportingDetectorFactorySelector	1	0	No
deref.UnconditionalValueDerefSet	8	64	No	plan.SingleDetectorFactorySelector	1	0	No
detect.AppendingToAnObjectOutputStream	1	0	No	Plugin	15	155	No
detect.AtomicityProblem	2	0	No	PluginLoader	18	311	No
detect.BadlyOverriddenAdapter	1	0	No	ProgramPoint	6	17	No
detect.BadResultSetAccess	2	0	No	Project	23	297	No
detect.BadSyntaxForRegularExpression	1	0	No	ProjectPackagePrefixes	3	21	No
detect.BadUseOfReturnValue	2	0	No	ProjectStats	14	159	No
detect.BooleanReturnNull	5	50	No	PropertyBundle	1	0	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
detect.BuildNonNullAnnotationDatabase	1	0	No	props.WarningProperty	1	0	No
detect.BuildNonnullReturnDatabase	11	464	No	props.WarningPropertySet	7	658	No
detect.BuildObligationPolicyDatabase	12	508	No	ResourceCollection	3	16	No
detect.BuildUnconditionalParamDeref Database	20	1532	No	ResourceTrackingDetecto	11	963	No
detect.CalledMethods	4	3	No	SAXBugCollectionHandler	12	105	Yes
detect.CheckImmutableAnnotation	3	66	No	SelfCalls	7	308	No
detect.CheckRelaxingNullnessAnnotation	4	6	No	SortedBugCollection	16	215	No
detect.CheckTypeQualifiers	4	16	No	SourceLineAnnotation	58	648	No
detect.CloneIdiom	10	94	No	sourceViewer.HighlightInformation	2	23	No
detect.ComparatorIdiom	6	75	No	sourceViewer.JavaScanner	1	0	No
detect.ConfusedInheritance	3	7	No	sourceViewer.JavaSourceDocument	6	130	No
detect.ConfusionBetweenInheritedAnd OuterMethod	4	2	No	sourceViewer.NavigableTextPane	2	1	No
detect.CrossSiteScripting	2	1	No	sourceviewer.NumberedEditor Kit	1	0	No
detect.DefaultEncodingDetector	5	7	No	sourceViewer.NumberedParagraphView	2	33	No
detect.DoInsideDoPrivileged	4	10	No	StringAnnotation	5	107	No
detect.DontCatchIllegalMonitorStateException	2	0	No	SuppressionMatcher	2	0	No
detect.DontIgnoreResultOfPutIfAbsent	12	448	No	SwitchHandler	7	186	No
detect.DroppedException	8	78	No	SystemPropertie	1	0	No
detect.DumbMethodInvocations	8	1562	No	SystemProperties	118	508	No
detect.DumbMethods	22	1588	No	TextUIBugReporter	1	0	No
detect.DuplicateBranches	7	230	No	type.BottomType	2	4	No
detect.ExplicitSerialization	9	81	No	type.DoubleExtraType	1	0	No
detect.FieldItemSummary	10	21	No	type.ExceptionObjectType	3	1	No
detect.FinalizerNullsFields	1	0	No	type.ExceptionSet	7	175	No
detect.FindBadCast2	26	1574	No	type.ExceptionSetFactory	3	1	No
detect.FindBadForLoop	1	0	No	type.FieldStoreTypeDatabase	1	0	No
detect.FindBugsSummaryStats	4	28	No	type.LongExtraType	1	0	No
detect.FindDeadLocalStores	20	1381	No	type.NullType	6	5	No
detect.FindDoubleCheck	3	6	No	type.StandardTypeMerger	10	413	No
detect.FindFieldSelfAssignment	1	0	No	type.TopType	5	11	No
detect.FindFinalizeInvocations	3	73	No	type.TypeAnalysis	26	285	No
detect.FindFloatEquality	5	13	No	type.TypeDataflow	21	339	No
detect.FindHEmismatch	13	478	No	type.TypeFrame	21	195	No
detect.FindInconsistentSync2	31	4327	No	type.TypeFrameModelingVisitor	28	221	Yes
detect.FindJSR166LockMonitorenter	11	249	No	type.TypeMerger	2	1	No
detect.FindLocalSelfAssignment2	5	16	No	TypeAnnotation	11	221	No
detect.FindMaskedFields	12	300	No	ui2.FilterListener	1	0	No
detect.FindMismatchedWaitOrNotify	10	219	No	updates.PluginUpdateListener	1	0	No
detect.FindNakedNotify	2	3	No	updates.UpdateCheckCallback	1	0	No
detect.FindNonShortCircuit	3	1	No	updates.UpdateChecker	10	152	Yes
detect.FindNullDeref	49	7590	No	util.Bag	1	0	No
detect.FindNullDerefsInvolvingNonShortCircuitEvaluation	11	238	No	util.ClassName	36	323	Yes

Class	DC	BC	Unit test	Class	DC	BC	Unit test
detect.FindOpenStream	17	3125	No	util.ClassPathUtil	2	20	No
detect.FindPuzzlers	15	212	No	util.DualKeyHashMap	6	81	No
detect.FindRefComparison	27	1869	No	util.EditDistance	2	0	No
detect.FindReturnRef	7	20	No	util.JavaWebStart	2	14	No
detect.FindRunInvocations	3	8	No	util.LaunchBrowser	1	0	No
detect.FindSelfComparison	10	313	No	util.MapCache	4	88	No
detect.FindSelfComparison2	13	388	No	util.MultiMap	4	31	No
detect.FindSleepWithLockHeld	8	284	No	util.SplitCamelCaseIdentifier	2	1	Yes
detect.FindSpinLoop	2	13	No	util.StringMatcher	1	0	No
detect.FindSqlInjection	12	1783	Yes	util.Strings	4	9	Yes
detect.FindTwoLockWait	7	154	No	util.SubtypeTypeMatcher	4	85	No
detect.FindUncalledPrivateMethods	4	10	No	util.TopologicalSort	6	46	No
detect.FindUnconditionalWait	2	3	No	util.TypeMatcher	1	0	No
detect.FindUninitializedGet	11	151	No	util.Util	26	396	No
detect.FindUnrelatedTypesInGenericContainer	33	2446	No	Version	4	79	No
detect.FindUnreleasedLock	6	808	No	visitclass.AnnotationVisitor	2	7	No
detect.FindUnsatisfiedObligation	5	49	No	visitclass.DismantleBytecode	15	209	Yes
detect.FindUnsyncGet	3	4	No	visitclass.LVTHelper	2	8	No
detect.FindUselessControlFlow	2	0	No	visitclass.PreorderVisito	1	0	No
detect.FindUseOfNonSerializableValue	11	344	No	visitclass.PreorderVisitor	23	104	Yes
detect.FormatStringChecker	5	648	No	visitclass.Util	5	10	No
detect.FunctionsThatMightBeMistakenForProcedure	10	436	No	vna.AvailableLoad	4	0	No
detect.HugeSharedStringConstants	1	0	No	vna.LoadedFieldSet	2	2	No
detect.IDivResultCastToDouble	5	27	No	vna.MergeTree	6	137	No
detect.IncompatMask	2	3	Yes	vna.ValueNumber	29	143	No
detect.InconsistentAnnotations	9	711	No	vna.ValueNumberAnalysis	18	613	No
detect.InefficientToArray	5	18	No	vna.ValueNumberAnalysisFeatures	1	0	No
detect.InfiniteLoop	1	0	No	vna.ValueNumberCache	2	35	No
detect.InfiniteRecursiveLoop	6	34	No	vna.ValueNumberDataflow	23	283	No
detect.InheritanceUnsafeGetResource	4	24	No	vna.ValueNumberFactory	7	8	No
detect.InitializationChain	1	0	No	vna.ValueNumberFrame	36	211	No
detect.InitializeNonnullFieldsInConstructor	4	5	No	vna.ValueNumberFrameModelingVisitor	10	665	No
detect.InstanceFieldLoadStreamFactory	4	11	No	vna.ValueNumberSourceInfo	14	311	No
detect.InstantiateStaticClass	6	19	No	xml.OutputStreamXMLOutput	3	27	No
detect.IntCast2LongAsInstant	4	9	No	xml.XMLAttributeList	14	791	No
detect.InvalidJUnitTest	5	6	No	xml.XMLOutput	15	879	No
detect.IOStreamFactory	5	18	No	xml.XMLOutputUtil	3	0	No
detect.IteratorIdioms	6	58	No	xml.XMLUtil	2	6	No

Appendix Table II: Centrality metrics values for JabRef

Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
autocompleter.AbstractAutoCompleter	4	25	Yes	jabref.autocompleter.NameFieldAutoCompleter	1	0	No
autocompleter.AutoCompleterFactory	1	0	No	jabref.BaseAction	1	0	No
autocompleter.CrossrefAutoCompleter	1	0	No	jabref.BasePanel	57	368	No

Class	DC	BC	Unit test	Class	DC	BC	Unit test
autocompleter.DefaultAutoCompleter	1	0	No	jabref.BibtexDatabase	25	168	Yes
autocompleter.EntireFieldAutoCompleter	1	0	No	jabref.BibtexEntry	52	5773	Yes
autocompleter.NameFieldAutoCompleter	2	11	No	jabref.BibtexEntryType	22	143	No
collab.Change	2	75	No	jabref.BibtexFields	13	915	No
collab.ChangeDisplayDialog	1	0	No	jabref.BrowseAction	1	0	No
collab.ChangeScanner	11	491	No	jabref.CallBack	6	10	No
collab.EntryChange	3	35	No	jabref.CrossRefEntryComparator	1	0	No
collab.FileUpdateMonitor	4	1	No	jabref.DatabaseChangeEvent	3	281	No
collab.FileUpdatePanel	2	6	No	jabref.DatabaseChangeListener	1	0	No
collab.MetaDataChange	2	0	No	jabref.DuplicateCheck	3	8	Yes
core.generated._JabRefPlugin	1	0	No	jabref.EntryComparator	3	104	No
core.JabRefPlugin	1	0	No	jabref.EntryEditor	19	182	No
date.DatePickerButton	1	0	No	jabref.EntryEditorPrefsTab	2	0	No
EntryEditorTabList	1	0	No	jabref.EntryEditorTab	10	422	No
export.AutoSaveManager	8	33	No	jabref.EntryEditorTabList	1	0	No
export.CustomExportDialog	4	3	No	jabref.EntrySorter	3	283	No
export.CustomExportList	4	1	No	jabref.EntryTypeDialog	4	12	No
export.ExportCustomizationDialog	5	17	No	jabref.ExternalTab	6	127	No
export.ExportFileFilter	1	0	No	jabref.FieldComparator	5	431	No
export.ExportFormats	16	1215	No	jabref.FieldEditor	4	559	No
export.FieldFormatter	1	0	No	jabref.FieldEditorFocusListene	1	0	No
export.FileActions	12	386	No	jabref.FieldNameLabel	1	0	No
export.IExportFormat	2	280	No	jabref.FieldTextArea	3	289	No
export.LatexFieldFormatter	5	25	No	jabref.FieldTextField	4	6	No
export.ModsExportFormat	3	10	No	jabref.FieldTextMenu	1	0	No
export.MSBibExportFormat	1	0	No	jabref.FileHistory	4	0	No
export.MySQLExport	2	39	No	jabref.FileTab	3	0	No
export.OOCalcDatabase	7	694	No	jabref.FindUnlinkedFilesDialog	4	31	No
export.OpenDocumentRepresentation	7	694	No	jabref.FontSelectorDialog	1	0	No
export.OpenDocumentSpreadsheetCreator	1	0	No	jabref.GeneralRenderer	4	5	No
export.OpenOfficeDocumentCreator	1	0	No	jabref.GeneralTab	3	0	No
export.PluginBasedExportFormat	2	0	No	jabref.GlobalFocusListener	1	0	No
export.PostgreSQLExport	2	39	No	jabref.Globals	110	118	No
export.SaveAllAction	4	0	No	jabref.gui.FileListEditor	1	0	No
export.SaveDatabaseAction	16	326	No	jabref.GUIGlobals	37	1161	No
export.SaveSession	8	520	No	jabref.IdComparator	1	0	No
export.VerifyingWriter	3	1	No	jabref.ImportSettingsTab	2	0	No
exporter.DBExporter	10	691	No	jabref.IncrementalSearcher	2	11	No
exporter.MySQLExporter	1	0	No	jabref.JabRef	14	203	No
exporter.PostgreSQLExporter	1	0	No	jabref.JabRefFrame	51	311	No
external.AutoSetExternalFileForEntries	10	142	No	jabref.JabRefPreferences	98	967	No
external.DownloadExternalFile	2	0	No	jabref.JTextAreaWithHighlighting	2	0	No
external.ExternalFileMenuItem	3	3	No	jabref.MarkEntriesAction	9	52	No
external.ExternalFilePanel	4	47	No	jabref.MergeDialog	4	280	No
external.ExternalFileType	9	260	No	jabref.MergeDialog_ok_actionAdapter	1	0	No
external.ExternalFileTypeEditor	2	2	No	jabref.MetaData	20	566	No

Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
external.FileLinksUpgradeWarning	2	24	No	jabref.MnemonicAwareAction	1	0	No
external.PushToApplication	3	1	No	jabref.NameFormatterTab	4	2	No
external.PushToApplicationAction	7	207	No	jabref.OpenFileFilter	1	0	No
external.PushToApplicationButton	8	94	No	jabref.PrefsDialog3	4	280	No
external.PushToEmacs	2	0	No	jabref.PrefsTab	1	0	No
external.PushToLatexEditor	2	0	No	jabref.PreviewPanel	7	77	No
external.PushToLyx	4	1	No	jabref.PreviewPrefsTab	4	8	No
external.PushToTeXstudio	3	0	No	jabref.RightClickMenu	17	667	No
external.PushToVim	2	0	No	jabref.SearchManager2	15	114	No
external.PushToWinEdt	3	0	No	jabref.SearchRule	1	0	No
external.SynchronizeFileField	1	0	No	jabref.SearchRuleSet	2	280	No
external.WriteXMPAction	9	227	No	jabref.SearchTextListener	1	0	Yes
external.WriteXMPEntryEditorAction	2	0	No	jabref.SidePane	2	0	No
format.AuthorAbbreviator	2	280	No	jabref.SidePaneComponent	2	1	No
format.AuthorFirstFirst	1	0	Yes	jabref.SidePaneManager	7	109	No
format.AuthorLastFirst	1	0	No	jabref.SplashScreen	1	0	No
format.AuthorLastFirstAbbrCommas	1	0	Yes	jabref.TabLabelPattern	4	35	No
format.AuthorLastFirstAbbreviator	1	0	Yes	jabref.TableColumnsTab	3	0	No
format.Authors	3	239	No	jabref.TablePrefsTab	3	24	No
format.CreateDocBookAuthors	2	280	No	jabref.TransferableBibtexEntry	1	0	No
format.CreateDocBookEditors	1	0	No	jabref.Util	52	550	Yes
format.DOICheck	1	0	Yes	jabref.Worker	6	10	No
format.DOIStrip	1	0	No	jabref.XmpPrefsTab	2	0	No
format.FileLink	4	11	No	journals.AbbreviateAction	4	8	No
format.GetOpenOfficeType	2	1	No	journals.JournalAbbreviations	8	110	No
format.IfPlural	1	0	No	journals.ManageJournalsAction	3	3	No
format.RemoveBrackets	2	1	No	journals.ManageJournalsPanel	7	20	No
format.RemoveLatexCommands	1	0	No	journals.UnabbreviateAction	4	8	No
format.RemoveWhitespace	2	1	No	label.HandleDuplicateWarnings	3	40	No
format.ResolvePDF	2	0	Yes	label.LabelMaker	2	0	No
format.RisAuthors	1	0	No	labelPattern.LabelPattern	5	9	No
format.RisKeywords	1	0	No	labelPattern.LabelPatternPanel	4	21	No
format.WrapContent	1	0	No	labelPattern.LabelPatternUtil	10	685	Yes
format.WrapFileLinks	4	927	No	labelPattern.ResolveDuplicateLabelDialog	5	14	No
format.XMLChars	1	0	No	labelPattern.SearchFixDuplicateLabels	7	114	No
groups.AbstractGroup	4	4	No	layout.AbstractParamLayoutFormatter	5	840	No
groups.AllEntriesGroup	1	0	No	layout.format.Replace	1	0	No
groups.GroupSelector	8	96	No	layout.Layout	5	17	Yes
groups.GroupsPrefsTab	2	0	No	layout.LayoutEntry	13	112	Yes
groups.GroupsTree	4	11	No	layout.LayoutFormatter	1	0	No
groups.GroupTreeCellRenderer	6	99	No	layout.LayoutHelper	12	779	No
groups.GroupTreeNode	7	20	No	layout.ParamLayoutFormatter	1	0	No
gui.AutoCompleteListener	4	35	No	layout.WSITools	2	49	No
gui.CleanUpAction	4	0	No	mods.MODSDatabase	3	28	No
gui.ColorSetupPanel	3	0	No	mods.MODSEntry	5	235	No
gui.DatabasePropertiesDialog	4	4	No	mods.PageNumbers	2	1	No
gui.DragDropPopupPane	2	0	No	mods.PersonName	4	72	No
gui.FileDialogs	6	561	No	msbib.MSBibDatabase	3	28	No
gui.FileListEditor	8	156	No	msbib.MSBibEntry	5	235	No
gui.FileListEntry	6	15	No	oo.OpenOfficePanel	8	106	No
gui.FileListEntryEditor	8	78	No	plugin.ManagePluginsDialog	4	45	No
gui.FileListTableModel	10	210	No	plugin.PluginCore	8	378	No
gui.GlazedEntrySorter	1	0	No	plugin.PluginInstaller	7	98	No

Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
gui.IsMarkedComparator	1	0	No	plugin.PluginInstallerAction	2	0	No
gui.MainTable	12	154	No	ritopt.BooleanOption	1	0	No
gui.MainTableFormat	13	1316	No	ritopt.Option	1	0	No
gui.MainTableSelectionListener	12	200	No	ritopt.OptionModule	2	280	No
gui.PreventDraggingJTableHeader	1	0	No	ritopt.Options	2	558	No
gui.SortTabsAction	1	0	No	ritopt.StringOption	1	0	No
help.HelpAction	5	10	No	search.BasicSearch	2	280	Yes
help.HelpContent	2	0	No	search.SearchExpression	2	29	No
help.HelpDialog	5	44	No	search.SearchExpressionLexer	1	0	No
importer.DbImportAction	8	1180	No	search.SearchExpressionParser	3	7	No
importer.DBImporter	1	0	No	search.SearchMatcher	1	0	No
importer.MySQLImporter	3	15	No	sf.jabref.EntryEditor	1	0	No
imports.ACMPortalFetcher	1	0	No	specialfields.Priority	3	1	No
imports.ADSFetcher	1	0	No	specialfields.Quality	5	12	No
imports.AppendDatabaseAction	12	479	No	specialfields.Rank	7	46	No
imports.BibtexParser	8	211	Yes	specialfields.RankCompact	3	0	No
imports.CheckForNewEntryTypesActio n	1	0	No	specialfields.Relevance	5	12	No
imports.CustomImportList	3	85	No	specialfields.SpecialField	3	2	No
imports.EntryFetcher	1	0	No	specialfields.SpecialFieldAction	5	23	No
imports.EntryFromFileCreator	2	0	No	specialfields.SpecialFieldDataba seChangeListener	3	279	No
imports.EntryFromFileCreatorManager	4	76	Yes	specialfields.SpecialFieldMenuA ction	3	0	No
imports.EntryFromPDFCreator	1	0	Yes	specialfields.SpecialFieldsUtils	7	211	No
imports.FieldContentParser	1	0	No	specialfields.SpecialFieldUpdat eListener	1	0	No
imports.GeneralFetcher	4	280	Yes	specialfields.SpecialFieldValue	8	53	No
imports.IEEEXploreFetcher	1	0	No	sql.DBConnectDialog	4	11	No
imports.ImportFormat	1	0	No	sql.DBExporterAndImporterFac tory	6	641	No
imports.ImportFormatReader	5	284	No	sql.DBImporterExporter	2	4	No
imports.ImportFormats	1	0	No	sql.DBStrings	6	295	No
imports.INSPIREFetcher	1	0	No	sql.SQLUtil	6	106	No
imports.OAI2Fetcher	1	0	Yes	undo.CountingUndoManager	9	14	No
imports.OpenDatabaseAction	16	473	No	undo.NamedCompound	1	0	No
imports.ParserResult	7	291	No	undo.UndoableFieldChange	1	0	No
imports.PdfXmpImporter	1	0	No	undo.UndoableInsertEntry	3	0	No
imports.PostOpenAction	1	0	No	util.CaseChangeMenu	2	280	No
imports.SPIRESFetcher	1	0	No	util.CaseChanger	1	0	Yes
jabref.AbstractWorker	3	3	No	util.ErrorConsole	2	0	No
jabref.AdvancedTab	4	8	No	util.TBuildInfo	1	0	No
jabref.AppearancePrefsTab	3	0	No	util.TXMLReader	1	0	No
jabref.AuthorList	16	2921	Yes	util.Util	1	0	Yes

Appendix Table III: Centrality metrics values for Dependency Finder

Class	DC	BC	Unit test	Class	DC	BC	Unit test
dependency.Node	60	1816	Yes	impl.FieldRef_info	3	19	No
dependency.Printer	53	2033	No	impl.Integer_info	3	180	No
dependency.VisitorBase	52	1283	No	impl.Method_info	3	10	Yes
dependency.RegularExpressionSelectio nCriteria	50	2500	Yes	impl.Signature_attribute	3	9	Yes
dependency.TextPrinter	48	924	Yes	impl.String_info	3	11	No
classreader.Visitor	45	4105	No	classreader.Annotation	2	7	No
dependency.FeatureNode	40	478	Yes	classreader.AttributeType	2	0	No

Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
dependency.Visitor	38	386	No	classreader.Class_info	2	7	No
impl.ConstantPool	35	1658	No	classreader.ClassfileScanner	2	0	Yes
dependency.ClassNode	34	1243	Yes	classreader.ConstantPoolEntry	2	0	No
classreader.VisitorBase	31	2911	Yes	classreader.ExceptionHandler	2	7	No
dependency.TraversalStrategy	30	127	No	classreader.LoadListenerVisitor Adapter	2	1	Yes
dependency.MetricsGatherer	29	778	Yes	classreader.LocalVariableTable _attribute	2	0	No
gui.StatusLine	29	918	Yes	classreader.TransientClassfileLo ader	2	180	Yes
dependency.NodeFactory	27	214	Yes	commandline.CommandLineSw itch	2	0	No
dependency.CodeDependencyCollecto r	25	180	Yes	commandline.ParameterStrate gy	2	0	No
dependency.GraphSummarizer	25	137	Yes	commandline.Printer	2	180	No
dependency.SelectionCriteria	25	75	No	commandline.SingleValueSwitc h	2	2	Yes
classreader.ClassfileLoaderEventSourc e	24	355	Yes	commandline.ToggleSwitch	2	2	Yes
dependency.GraphCopier	22	57	Yes	dependency.ClosureLayerSelect or	2	5	No
gui.DependencyFinder	22	2673	No	dependency.TransitiveClosure	2	11	Yes
classreader.ZipClassfileLoader	21	32	Yes	gui.ClosureQueryAction	2	0	No
dependency.PackageNode	20	23	Yes	gui.DependencyQueryAction	2	0	No
classreader.DirectoryClassfileLoader	19	26	Yes	gui.MetricsQueryAction	2	0	No
dependency.ClosureStartSelector	17	41	Yes	gui.MetricsTableModel	2	180	No
impl.Feature_info	17	46	No	gui.NewDependencyGraphActi on	2	0	No
classreader.Monitor	15	86	Yes	impl.AnnotationDefault_attribu te	2	72	Yes
impl.Class_info	15	146	Yes	impl.ArrayElementValue	2	72	Yes
impl.Classfile	15	236	Yes	impl.BooleanConstantElementV alue	2	0	Yes
impl.Code_attribute	15	121	yes	impl.ElementValueType	2	180	No
classreader.JarClassfileLoader	14	5	Yes	impl.InnerClasses_attribute	2	0	Yes
impl.Instruction	14	209	Yes	impl.IntegerConstantElementV alue	2	0	Yes
impl.UTF8_info	14	142	No	impl.LocalVariableTable_attribu te	2	0	Yes
gui.StatusLineUpdater	13	110	No	impl.LocalVariableTypeTable_at tribute	2	0	Yes
classreader.ClassfileLoader	12	24	Yes	impl.StringConstantElementVal ue	2	4	Yes
classreader.ClassfileLoaderDecorator	12	3	No	classreader.AnnotationDefault_ attribute	1	0	No
dependency.SelectiveTraversalStrateg y	12	0	Yes	classreader.ArrayElementValue	1	0	No
dependency.XMLPrinter	12	241	Yes	classreader.ClassElementValue	1	0	No
impl.LocalVariable	12	167	Yes	classreader.ClassfileFactory	1	0	No
impl.NameAndType_info	11	56	No	classreader.Code_attribute	1	0	No
impl.MethodRef_info	10	124	No	classreader.ElementValue	1	0	No
classreader.DirectoryExplorer	9	2	Yes	classreader.ElementValuePair	1	0	No
dependency.ClosureSelector	9	23	No	classreader.ElementValueType	1	0	No
dependency.ClosureStopSelector	9	31	Yes	classreader.EnumElementValue	1	0	No
dependency.DecoratorTraversalStrate gy	9	0	No	classreader.Exceptions_attribut e	1	0	No
dependencyfinder.VerboseListenerBas e	9	97	No	classreader.FieldRef_info	1	0	No
impl.ExceptionHandler	9	36	No	classreader.InnerClass	1	0	No

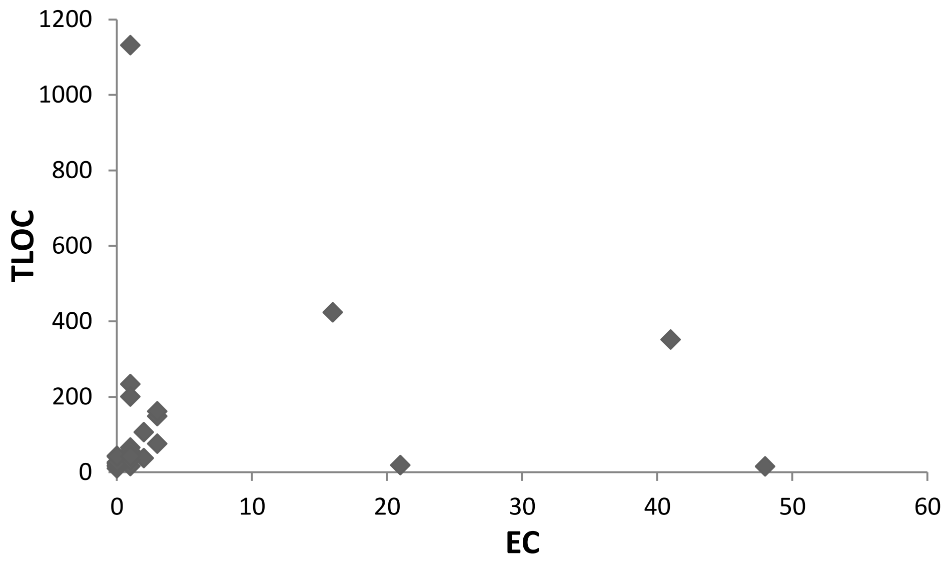
Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
impl.Exceptions_attribute	9	54	Yes	classreader.InnerClasses_attribute	1	0	No
impl.LineNumber	9	50	No	classreader.InterfaceMethodRef_info	1	0	No
impl.SourceFile_attribute	9	54	Yes	classreader.LineNumber	1	0	No
text.Hex	9	3	Yes	classreader.LineNumberTable_attribute	1	0	No
classreader.LoadEvent	8	302	No	classreader.LoadListene	1	0	No
impl.AttributeFactory	8	187	Yes	classreader.LocalVariable	1	0	No
impl.AttributeType	8	91	No	classreader.LocalVariableType	1	0	No
impl.FeatureRef_info	8	0	No	classreader.LocalVariableTypeTable_attribute	1	0	Yes
classreader.ClassNameHelper	7	403	Yes	classreader.MethodRef_info	1	0	No
dependency.MetricsReport	7	5	No	classreader.ModifiedOnlyDispatcher	1	0	Yes
impl.LocalVariableType	7	90	No	classreader.NameAndType_info	1	0	No
classreader.DescriptorHelper	6	25	Yes	classreader.Parameter	1	0	No
commandline.CommandLine	6	1734	Yes	classreader.RuntimeAnnotations_attribute	1	0	No
dependency.TransitiveClosureEngine	6	15	Yes	classreader.RuntimeParameterAnnotations_attribute	1	0	No
gui.DependencyExtractAction	6	104	No	commandline.NullParameterStrategy	1	0	Yes
gui.RefreshDependencyGraphAction	6	94	No	dependency.DecoratorTraversalStrateg	1	0	No
gui.VerboseListener	6	7	No	dependencyfinder.gui.StatusLineUpdater	1	0	No
impl.ConstantValue_attribute	6	95	Yes	dependencyfinder.Version	1	0	No
impl.InnerClass	6	53	No	gui.AboutAction	1	0	No
classreader.Classfile	5	76	No	gui.AdvancedQueryPanelAction	1	0	No
classreader.Method_info	5	9	No	gui.DependencyExtractActio	1	0	No
classreader.DescriptorIterator	4	24	No	gui.DependencyFinde	1	0	No
classreader.LoadListenerBase	4	26	No	gui.RefreshDependencyGraphActio	1	0	No
commandline.CommandLineUsage	4	534	Yes	gui.SimpleQueryPanelAction	1	0	No
commandline.Visitor	4	356	No	gui.StatusLineUpdate	1	0	No
gui.AllQueriesAction	4	4	No	impl.Code_attribut	1	0	No
impl.Annotation	4	4	Yes	impl.ConstantElementValue	1	0	No
impl.ClassElementValue	4	80	Yes	impl.Custom_attribute	1	0	Yes
impl.ElementValueFactory	4	361	Yes	impl.Deprecated_attribute	1	0	Yes
impl.ElementValuePair	4	446	Yes	impl.EnclosingMethod_attribut	1	0	No
impl.EnumElementValue	4	80	Yes	impl.Field_info	1	0	No
classreader.Attribute_info	3	2	No	impl.Float_info	1	0	No
classreader.ClassfileLoaderAction	3	0	No	impl.InterfaceMethodRef_info	1	0	Yes
classreader.ClassfileLoaderDispatcher	3	180	No	impl.LineNumberTable_attribute	1	0	Yes
classreader.Field_info	3	9	No	impl.Long_info	1	0	No
classreader.GroupData	3	2	No	impl.LongConstantElementValue	1	0	Yes
classreader.Instruction	3	15	Yes	impl.Parameter	1	0	Yes
classreader.UTF8_info	3	3	No	impl.RuntimeInvisibleAnnotations_attribute	1	0	Yes
commandline.VisitorBase	3	1	No	impl.RuntimeVisibleAnnotations_attribut	1	0	No
dependency.CodeDependencyCollecto	3	1308	No	impl.RuntimeVisibleParameterAnnotations_attribute	1	0	Yes
gui.SaveFileAction	3	0	No	impl.Signature_attribut	1	0	No
impl.CodeIterator	3	0	No	impl.Synthetic_attribute	1	0	Yes
impl.EnclosingMethod_attribute	3	0	Yes	text.PrinterBuffer	1	0	Yes
				text.RegularExpressionParser	1	0	yes

Appendix Table IV: Centrality metrics values for MOEA

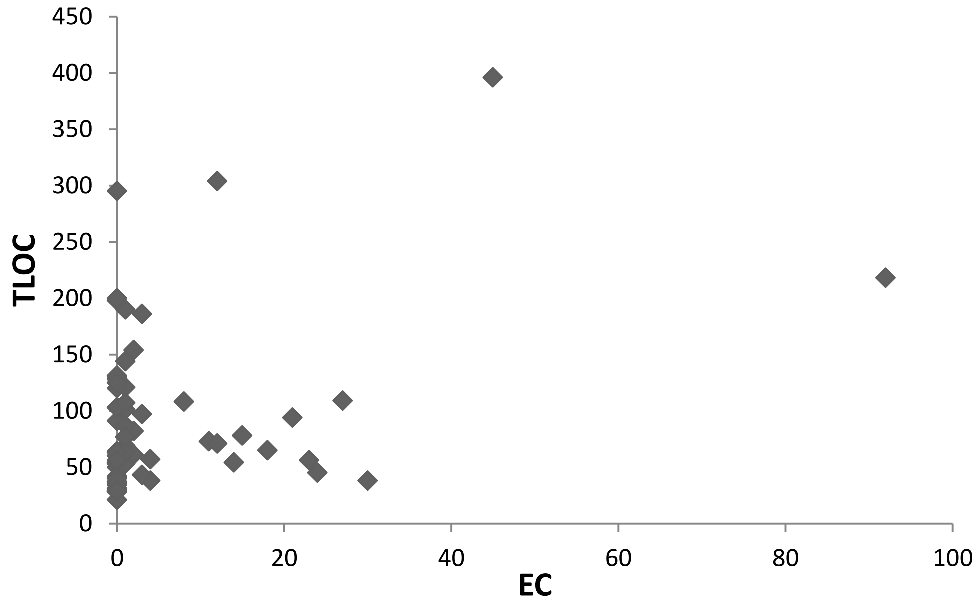
Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
AbstractAlgorithm	4	9	No	NormalizedIndicator	1	0	No
AbstractEvolutionaryAlgorithm	2	12	Yes	Normalizer	5	158	Yes
Accumulator	22	847	Yes	NSGAII	6	40	Yes
ActionFactory	16	65	No	ObjectiveComparator	2	0	Yes
AdaptiveMultimethodVariationCollect or	1	0	Yes	OperatorFactory	6	101	Yes
AdaptiveTimeContinuation	2	1	Yes	PaintHelper	5	3	No
AdaptiveTimeContinuationCollector	3	132	Yes	ParetoObjectiveComparator	15	64	Yes
AdditiveEpsilonIndicator	6	17	Yes	PeriodicAction	1	0	Yes
AggregateConstraintComparator	23	107	Yes	PM	9	24	Yes
Algorithm	6	159	Yes	Population	24	251	Yes
AlgorithmFactory	1	0	No	PopulationIO	6	151	Yes
AlgorithmProvider	1	0	No	PopulationSizeCollector	5	173	Yes
Analyzer	2	0	Yes	PRNG	13	157	Yes
ApproximationSetCollector	5	52	Yes	Problem	15	224	No
ApproximationSetPlot	9	25	No	ProblemBuilder	2	16	Yes
ApproximationSetViewer	6	17	No	ProblemFactory	4	175	No
AttachPoint	9	200	No	ProblemProvider	1	0	No
CEC2009	21	70	Yes	Problems	11	862	No
CF10	3	5	No	PropertiesProblems	1	0	Yes
CF2	3	5	No	RandomInitialization	7	33	Yes
CF3	3	5	No	RandomSearch	2	12	No
CF4	3	5	No	RankComparator	1	0	Yes
CF5	3	5	No	RealVariable	11	23	Yes
CF6	3	5	No	ResultKey	6	8	No
CF7	5	11	No	ResultPlot	9	46	No
CF8	3	5	No	RotatedProblem	8	173	Yes
CF9	3	5	No	RotatedProblems	3	20	Yes
ChainedComparator	11	24	Yes	RotationMatrixBuilder	2	5	Yes
Collector	3	4	Yes	SBX	9	26	No
CompoundVariation	4	9	Yes	Selection	2	0	No
Contribution	2	146	Yes	Settings	4	160	Yes
Controller	35	1234	No	ShapeFunctions	2	74	No
ControllerEvent	21	244	No	Shapes	3	360	No
ControllerListener	8	21	No	Solution	86	572	Yes
CrowdingComparator	2	0	Yes	SortedListModel	5	5	No
DiagnosticTool	25	544	No	Spacing	3	0	Yes
DifferentialEvolution	7	30	Yes	StandardAlgorithms	6	97	No
DifferentialEvolutionSelection	5	4	Yes	StandardProblems	1	0	Yes
DominanceComparator	13	26	No	StatisticalResultsViewer	2	1	No
DTLZ	4	11	No	TournamentSelection	6	8	Yes
DTLZ1	2	0	Yes	TransFunctions	2	73	No
DTLZ2	3	0	Yes	Transitions	2	214	No
DTLZ3	3	0	Yes	TypedProperties	3	4	Yes
DTLZ4	2	0	Yes	UF1	3	5	No
DTLZ7	2	0	Yes	UF10	3	5	No
ElapsedTimeCollector	3	21	Yes	UF11	4	6	Yes
EncodingUtils	48	1084	Yes	UF12	5	27	Yes
EpsilonBoxDominanceArchive	16	309	Yes	UF2	3	5	No
EpsilonBoxDominanceComparato	1	0	Yes	UF3	3	5	No
EpsilonBoxDominanceComparator	10	12	Yes	UF4	3	5	No
EpsilonBoxEvolutionaryAlgorithm	1	0	No	UF5	3	5	No
EpsilonBoxObjectiveComparator	21		Yes	UF6	3	5	No
EpsilonHelper	4	13	Yes	UF7	4	8	No
EpsilonMOEA	10	169	Yes	UF8	3	5	No

Class	DC	BC	Unit Test	Class	DC	BC	Unit Test
EpsilonProgressCollector	4	150	Yes	UF9	3	5	No
EvolutionaryAlgorithm	2	14	No	Variable	6	1	No
FastNondominatedSorting	5	7	Yes	Variation	6	11	No
FrameworkFunctions	1	0	No	Vector	1	0	Yes
GDE3	7	11	Yes	WFG	3	105	Yes
GenerationalDistance	2	0	Yes	WFG1	3	103	No
Hypervolume	4	81	Yes	WFG2	3	103	No
Indicator	1	0	No	WFG3	3	103	No
IndicatorCollector	7	265	Yes	WFG4	3	103	No
IndicatorUtils	9	111	Yes	WFG5	3	103	No
Initialization	3	3	No	WFG6	3	103	No
InstrumentedAlgorithm	4	29	No	WFG7	3	103	No
Instrumenter	5	32	Yes	WFG8	3	103	No
InvertedGenerationalDistance	2	0	Yes	WFG9	3	103	No
LinePlot	17	142	No	ZDT	1	0	Yes
Localization	13	58	Yes	ZDT1	2	0	No
LZ	2	0	Yes	ZDT2	2	0	No
Misc	2	3	No	ZDT3	2	0	No
MOEAD	7	167	Yes	ZDT4	2	0	No
NondominatedPopulation	36	1404	Yes	ZDT5	1	0	No
NondominatedSortingPopulation	3	3	Yes	ZDT6	2	0	No

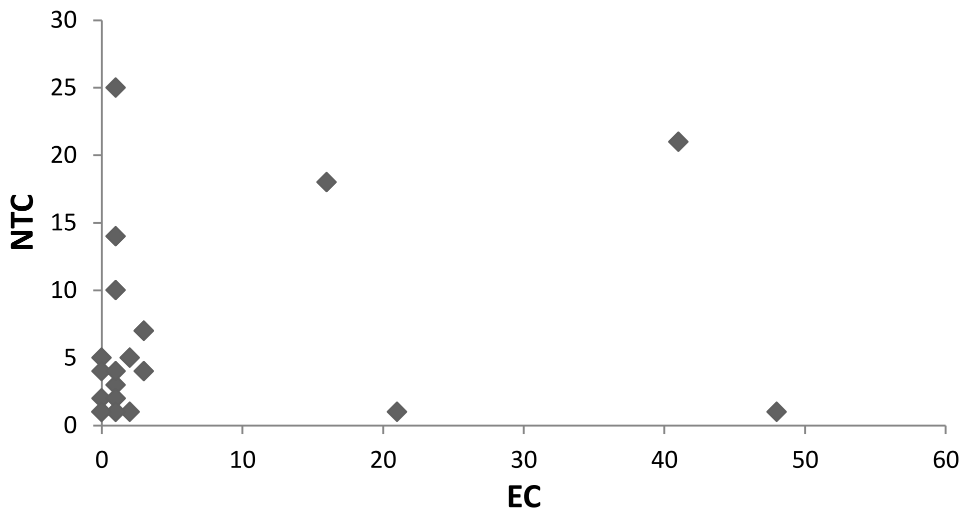
Appendix D: Relationships between Dynamic Metrics and Class Testability Metrics



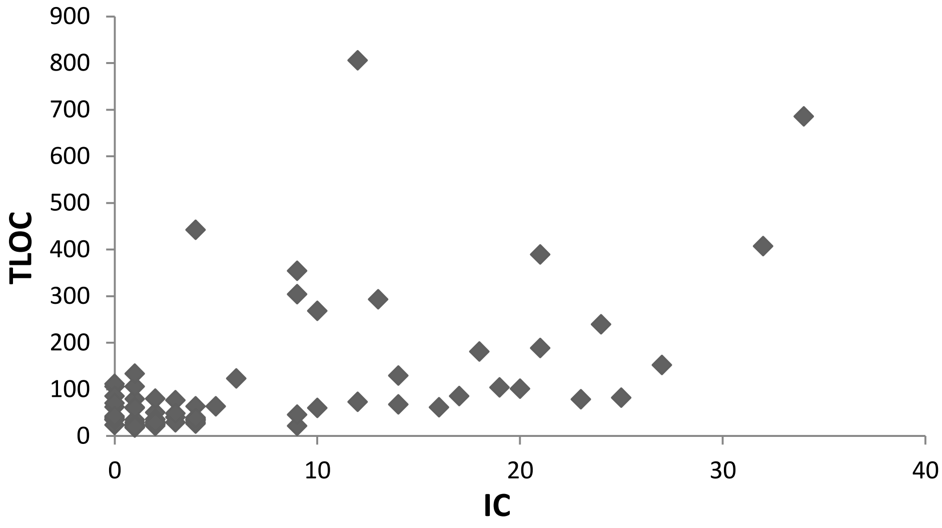
Appendix Figure I. Scatter lot of the relationship between EC and TLOC in *JabRef*



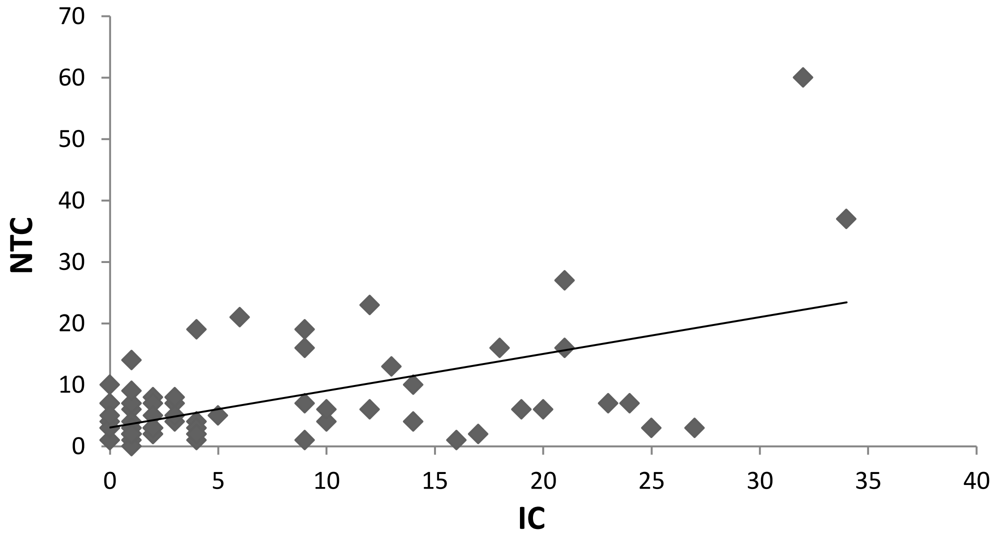
Appendix Figure II. Scatter plot of the relationship between EC and TLOC in MOEA



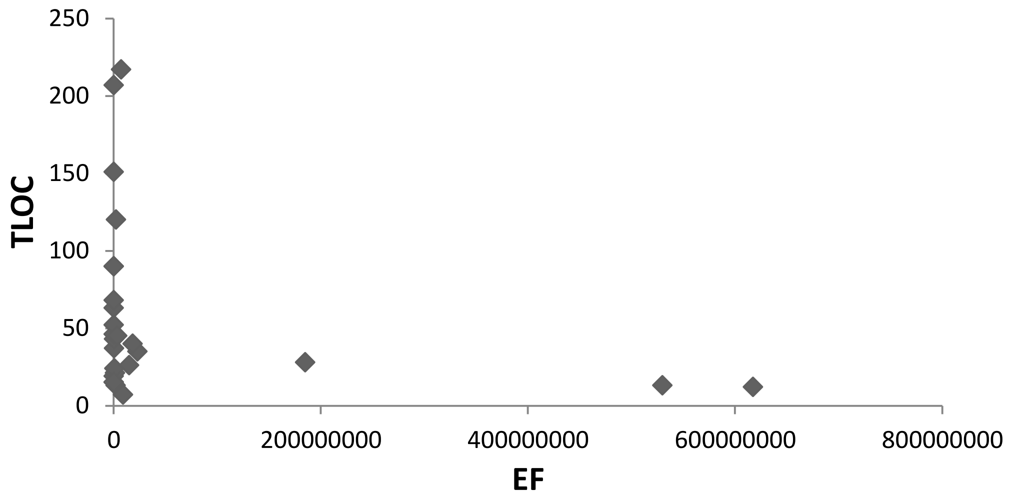
Appendix Figure III. Scatter plot of the relationship between EC and NTC in *JabRef*



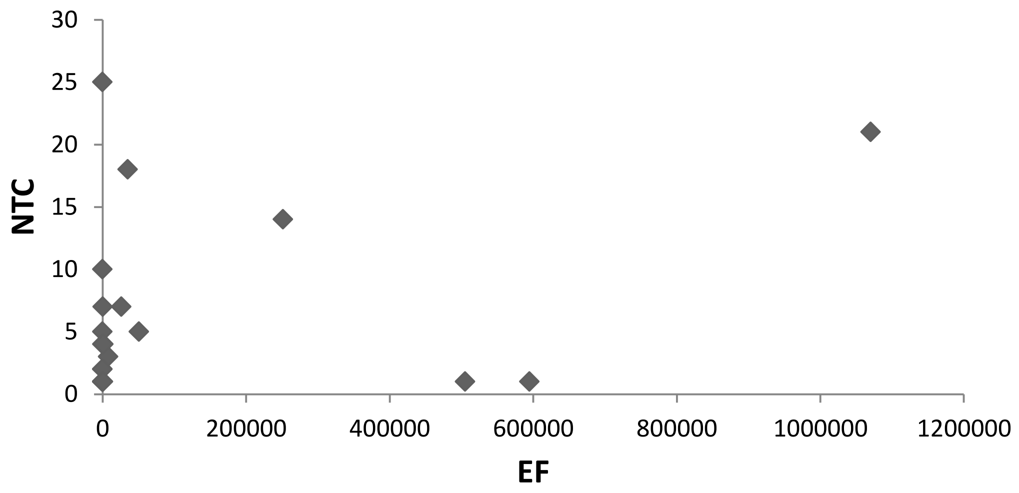
Appendix Figure IV. Scatter plot of the relationship between IC and TLOC in *Dependency Finder*



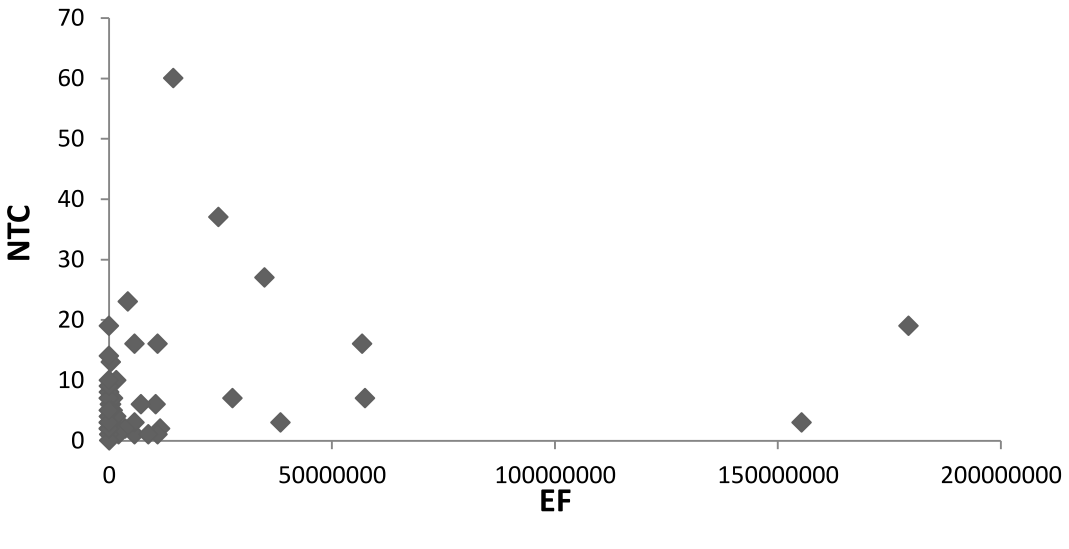
Appendix Figure V. Scatter plot of the relationship between IC and NTC in *Dependency Finder*



Appendix Figure VI. Scatter plot of the relationship between EF and TLOC in *FindBugs*



Appendix Figure VII. Scatter plot of the relationship between EF and NTC in *JabRef*



Appendix Figure VIII. Scatter plot of the relationship between EF and TLOC in *Dependency Finder*

Appendix E: Results of the Correlations between Test and Code Smells

Appendix Table V: Phi correlation test results for JFreeChart

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	0.22	0.06	0.04	-------	-------	**0.38**	**0.35**	0.14	0.21
	p	0.00*	0.27	0.46	-------	-------	**0.00***	**0.00***	0.03*	0.00*
Large Class	ϕ	0.03	0.04	-0.06	-------	-------	-0.09	0.27	0.18	0.11
	p	0.54	0.51	0.30	-------	-------	0.08	0.00*	0.00*	0.04*
Type Checking	ϕ	0.13	0.02	0.17	-------	-------	-0.04	0.13	-0.02	0.14
	p	0.01*	0.68	0.00*	-------	-------	0.51	0.02*	0.72	0.01*
Brain Class	ϕ	0.21	0.09	-0.06	-------	-------	**0.41**	**0.34**	0.16	0.17
	p	0.00*	0.10	0.25	-------	-------	**0.00***	**0.00***	0.00*	0.00*
Duplicated Code	ϕ	0.05	-0.01	-0.09	-------	-------	-0.08	0.24	0.12	0.27
	p	0.39	0.87	0.10	-------	-------	0.11	0.00*	0.02*	0.00*
Schizophrenic Class	ϕ	0.18	0.02	-0.01	-------	-------	-0.03	-0.03	0.18	0.04
	p	0.00*	0.75	0.82	-------	-------	0.61	0.58	0.00*	0.43
Data Class	ϕ	-0.03	-0.16	-0.02	-------	-------	-0.05	0.01	0.19	-0.08
	p	0.61	0.00*	0.68	-------	-------	0.35	0.93	0.00*	0.16
Data Clumps	ϕ	-0.07	0.08	-0.06	-------	-------	-0.13	**0.33**	0.17	0.05
	p	0.18	0.12	0.27	-------	-------	0.02*	**0.00***	0.00*	0.38

Appendix Table VI: Phi correlation test results for FindBugs

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	φ	-0.07	0.07	-0.09	**0.33**	-------	0.03	-------	-------	0.18
	p	0.67	0.68	0.60	**0.04***	-------	0.86	-------	-------	0.28
Large Class	φ	0.22	0.10	-0.13	-0.22	-------	0.04	-------	-------	0.26
	p	0.17	0.54	0.44	0.17	-------	0.80	-------	-------	0.11
Type Checking	φ	0.22	-0.05	0.14	0.13	-------	0.20	-------	-------	0.26
	p	0.17	0.78	0.39	0.42	-------	0.22	-------	-------	0.11
Brain Class	φ	0.19	0.14	0.24	0.02	-------	0.04	-------	-------	**0.31**
	p	0.24	0.39	0.15	0.90	-------	0.80	-------	-------	**0.05***
Duplicated Code	φ	0.26	-0.15	0.13	-0.08	-------	0.13	-------	-------	0.28
	p	0.11	0.36	0.43	0.64	-------	0.44	-------	-------	0.09
Schizophrenic Class	φ	-0.04	-0.18	-0.05	-0.09	-------	-0.11	-------	-------	-0.06
	p	0.81	0.26	0.77	0.60	-------	0.52	-------	-------	0.69
Data Class	φ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Data Clumps	φ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------

Appendix Table VII: Phi correlation test results for JMeter

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	0.11	0.01	-0.03	-0.01	0.25	0.23	**0.37**	0.08	0.04
	p	0.11	0.91	0.65	0.94	0.00*	0.00*	**0.00***	0.27	0.61
Large Class	ϕ	0.10	-0.05	-0.03	0.14	0.23	0.16	**0.35**	0.07	0.15
	p	0.16	0.53	0.63	0.05*	0.00*	0.03*	**0.00***	0.32	0.03*
Type Checking	ϕ	0.08	0.03	-0.02	0.22	-0.03	0.27	0.18	-0.03	0.05
	p	0.27	0.65	0.76	0.00*	0.70	0.00*	0.01*	0.66	0.48
Brain Class	ϕ	0.28	0.08	0.10	0.07	-0.05	0.21	0.20	**0.35**	0.01
	p	0.00*	0.29	0.16	0.31	0.47	0.00*	0.00*	**0.00***	0.90
Duplicated Code	ϕ	0.00	0.15	-0.06	0.02	0.03	0.05	0.22	0.00	0.12
	p	0.97	0.03*	0.42	0.84	0.72	0.49	0.00*	0.98	0.08
Schizophrenic Class	ϕ	-0.03	-0.01	-0.01	-0.05	-0.02	-0.05	0.24	-0.02	0.09
	p	0.72	0.87	0.86	0.53	0.83	0.49	0.00*	0.80	0.22
Data Class	ϕ	-0.03	-0.23	-0.02	0.06	-0.02	-0.06	0.20	-0.02	0.13
	p	0.68	0.00*	0.84	0.40	0.80	0.42	0.01*	0.77	0.08
Data Clumps	ϕ	-0.03	-0.10	-0.01	0.08	-0.02	-0.05	0.10	-0.02	0.00
	p	0.72	0.15	0.86	0.24	0.83	0.49	0.16	0.80	1.00

Appendix Table VIII: Phi correlation test results for JabRef

Smells		**Assertion-free**	**Assertion Roulette**	**Sensitive Equality**	**Mystery Test**	**Indirect Test**	**General Fixture**	**Eager Test**	**Lazy Test**	**Duplicated Code**
Feature Envy	ϕ	**0.43**	**-0.49**	-0.04	-0.06	-0.02	-0.06	**0.33**	-0.03	-0.09
	p	**0.00***	**0.00***	0.75	0.63	0.89	0.66	**0.01***	0.85	0.50
Large Class	ϕ	-0.09	0.08	-0.09	0.23	-0.04	0.07	0.28	-0.05	-0.09
	p	0.51	0.56	0.51	0.09	0.78	0.63	0.04*	0.69	0.39
Type Checking	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Brain Class	ϕ	0.03	-0.07	0.20	0.17	-0.06	0.07	**0.38**	-0.09	0.24
	p	0.82	0.63	0.13	0.20	0.66	0.60	**0.01***	0.52	0.08
Duplicated Code	ϕ	-0.06	0.18	**0.35**	-0.09	212.00	0.04	0.08	-0.12	**0.70**
	p	0.64	0.18	**0.01***	0.48	0.12	0.76	0.57	0.36	**0.00***
Schizophrenic Class	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Data Class	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Data Clumps	ϕ	-0.61	0.05	-0.06	0.16	-0.03	0.18	0.20	-0.04	0.08
	p	0.65	0.69	0.65	0.24	0.85	0.19	0.15	0.78	0.55

Appendix Table IX: Phi correlation test results for Commons Lang

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	0.22	0.09	0.14	-0.03	-0.02	-0.05	**0.33**	-0.03	0.06
	p	0.01*	0.33	0.13	0.77	0.86	0.58	**0.00***	0.77	0.51
Large Class	ϕ	-0.14	0.17	0.06	0.16	0.07	0.04	**0.31**	0.16	**0.46**
	p	0.13	0.06	0.52	0.09	0.45	0.65	**0.00***	0.09	**0.00***
Type Checking	ϕ	0.07	0.09	0.14	-0.03	-0.02	0.14	-0.05	-0.03	-0.08
	p	0.41	0.33	0.13	0.77	0.86	0.12	0.58	0.77	0.36
Brain Class	ϕ	0.20	0.25	0.17	-0.03	0.07	-0.01	**0.42**	0.06	0.22
	p	0.03	0.00*	0.06	0.73	0.45	0.88	**0.00***	0.49	0.02
Duplicated Code	ϕ	-0.16	0.13	-0.03	-0.05	0.05	0.09	0.20	0.12	**0.51**
	p	0.08	0.13	0.79	0.55	0.59	0.29	0.03*	0.18	**0.00***
Schizophrenic Class	ϕ	0.16	0.06	0.10	-0.02	-0.01	0.23	-0.04	-0.02	-0.06
	p	0.08	0.50	0.28	0.84	0.90	0.01*	0.70	0.84	0.52
Data Class	ϕ	0.03	0.11	0.06	-0.03	-0.02	0.10	-0.06	-0.03	0.02
	p	0.74	0.24	0.49	0.72	0.82	0.29	0.50	0.72	0.87
Data Clumps	ϕ	-0.07	-0.05	0.01	-0.03	-0.02	-0.05	-0.05	-0.03	0.06
	p	0.41	0.57	0.93	0.77	0.86	0.58	0.58	0.77	0.51

Appendix Table X: Phi correlation test results for Dependency Finder

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	0.04	0.09	0.00	-0.08	0.16	0.14	-0.08	-------	0.17
	p	0.55	0.18	1.00	0.26	0.02*	0.04*	0.24	-------	0.01*
Large Class	ϕ	0.00	0.14	-0.03	-0.03	0.14	-0.07	-0.08	-------	0.15
	p	0.99	0.04	0.68	0.63	0.04	0.27	0.26	-------	0.12
Type Checking	ϕ	-0.05	0.09	-0.04	0.02	-0.06	0.11	-0.07	-------	0.15
	p	0.50	0.19	0.57	0.80	0.41	0.11	0.32	-------	0.02*
Brain Class	ϕ	-0.02	0.14	0.01	-0.06	0.02	0.14	-0.07	-------	0.20
	p	0.74	0.03*	0.87	0.35	0.81	0.04*	0.31	-------	0.00*
Duplicated Code	ϕ	-0.06	0.07	-0.04	-0.01	-0.06	0.08	0.02	-------	0.14
	p	0.35	0.32	0.57	0.85	0.41	0.23	0.78	-------	0.04*
Schizophrenic Class	ϕ	-0.05	-0.11	0.09	-0.07	0.37	-0.12	-0.07	-------	0.04
	p	0.50	0.09	0.18	0.33	0.58	0.09	0.32	-------	0.56
Data Class	ϕ	0.23	0.07	0.13	0.25	0.07	0.07	0.24	-------	0.10
	p	0.00*	0.29	0.06	0.00*	0.31	0.33	0.00*	-------	0.13
Data Clumps	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------

Appendix Table XI: Phi correlation test results for MOEA

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	0.11	0.01	-0.03	-0.01	**0.25**	0.23	**0.37**	0.08	0.04
	p	0.11	0.91	0.65	0.94	**0.00***	0.00*	**0.00***	0.27	0.61
Large Class	ϕ	0.10	-0.05	-0.03	0.14	0.23	0.16	**0.35**	0.07	0.15
	p	0.16	0.53	0.63	0.05*	0.00*	0.03*	**0.00***	0.32	0.03*
Type Checking	ϕ	0.08	0.03	-0.02	0.22	-0.03	0.27	0.18	-0.03	0.05
	p	0.27	0.65	0.76	0.00*	0.70	0.00*	0.01*	0.66	0.48
Brain Class	ϕ	0.28	0.08	0.10	0.07	-0.05	0.21	0.20	**0.35**	0.01
	p	0.00*	0.29	0.16	0.31	0.47	0.00*	0.00*	**0.00***	0.90
Duplicated Code	ϕ	0.00	0.15	-0.06	0.02	0.03	0.05	0.22	0.00	0.12
	p	0.97	0.03*	0.42	0.84	0.72	0.49	0.00*	0.98	0.08
Schizophrenic Class	ϕ	-0.03	-0.01	-0.01	-0.05	-0.02	-0.05	0.24	-0.02	0.09
	p	0.72	0.87	0.86	0.53	0.83	0.49	0.00*	0.80	0.22
Data Class	ϕ	-0.03	-0.23	-0.02	0.06	-0.02	-0.06	0.20	-0.02	0.13
	p	0.68	0.00*	0.84	0.40	0.80	0.42	0.01*	0.77	0.08
Data Clumps	ϕ	-0.03	-0.10	-0.01	0.08	-0.02	-0.05	0.10	-0.02	0.00
	p	0.72	0.15	0.86	0.24	0.83	0.49	0.16	0.80	1.00

Appendix Table XII: Phi correlation test results for Barcode4J

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Feature Envy	ϕ	-0.10	0.13	-0.01	-------	-0.07	-------	0.38	**0.70**	-0.14
	p	.57	0.47	0.96	-------	0.70	-------	0.03	**0.00***	0.43
Large Class	ϕ	0.06	0.03	0.15	-------	-0.13	-------	0.19	**0.37**	0.32
	p	.76	0.85	0.41	-------	0.47	-------	0.30	**0.04***	0.07
Type Checking	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Brain Class	ϕ	0.02	0.27	0.06	-------	0.17	-------	**0.45**	0.34	0.26
	p	0.90	0.14	0.74	-------	0.34	-------	**0.01***	0.06	0.15
Duplicated Code	ϕ	-0.37	0.15	**0.42**	-------	0.01	-------	0.30	0.19	0.25
	p	0.04	0.41	**0.02***	-------	0.96	-------	0.10	0.29	0.17
Schizophrenic Class	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Data Class	ϕ	-------	-------	-------	-------	-------	-------	-------	-------	-------
	p	-------	-------	-------	-------	-------	-------	-------	-------	-------
Data Clumps	ϕ	-0.07	-0.37	-0.19	-------	-0.05	-------	-0.13	-0.03	-0.10
	p	0.70	0.04	0.29	-------	0.79	-------	0.48	0.85	0.58

Appendix F: Results of the Correlations between Test Smells

Appendix Table XIII: Results of the φ correlation coefficient analysis between test smells in JFreeChart

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	ϕ		-0.69	-0.02	-------	-------	**0.35**	0.07	0.09	0.17
	p		0.19	0.69	-------	-------	**0.00***	0.17	0.09	0.00*
Assertion Roulette	ϕ	-0.69		0.02	-------	-------	0.05	-0.05	-0.07	0.41
	p	0.19		0.65	-------	-------	0.31	0.36	0.19	0.44
Sensitive Equality	ϕ	-0.02	0.02		-------	-------	-0.04	0.43	-0.02	0.06
	p	0.69	0.65		-------	-------	0.47	-0.02	0.69	0.26
Mystery Test	ϕ	-------	-------	-------		-------	-------	-------	-------	-------
	p	-------	-------	-------		-------	-------	-------	-------	-------
Indirect Test	ϕ	-------	-------	-------	-------		-------	-------	-------	-------
	p	-------	-------	-------	-------		-------	-------	-------	-------
General Fixture	ϕ	**0.35**	0.05	-0.04	-------	-------		0.06	-0.05	0.18
	p	**0.00***	0.31	0.47	-------	-------		0.77	0.38	0.00*
Eager Test	ϕ	0.07	-0.05	-0.04	-------	-------	0.06		-0.05	0.17
	p	0.17	0.36	0.43	-------	-------	0.77		0.33	0.00*
Lazy Test	ϕ	0.09	-0.07	-0.02	-------	-------	-0.05	-0.05		-0.05
	p	0.09	0.19	0.69	-------	-------	0.38	0.33		0.33
Duplicated Code	ϕ	0.17	0.41	0.06	-------	-------	0.18	0.17	-0.05	
	p	0.00*	0.44	0.26	-------	-------	0.00*	0.00*	0.33	

Appendix Table XIV: Results of the φ correlation coefficient analysis between test smells in FindBugs

Smells		**Assertion-free**	**Assertion Roulette**	**Sensitive Equality**	**Mystery Test**	**Indirect Test**	**General Fixture**	**Eager Test**	**Lazy Test**	**Duplicated Code**
Assertion-free	φ		0.20	0.37	-0.12	0.20	0.12	**0.37**	**0.47**	0.26
	p		0.20	0.02	0.45	0.22	0.50	**0.02***	**0.00***	0.11
Assertion Roulette	φ	0.20		0.07	-0.19	-0.27	**-0.36**	0.07	-0.03	**0.36**
	p	0.20		0.68	0.24	0.09	**0.03***	0.68	0.88	**0.02***
Sensitive Equality	φ	0.37	0.07		0.09	0.12	0.03	0.27	**0.37**	-0.11
	p	0.02	0.68		0.59	0.45	0.86	0.09	**0.02***	0.48
Mystery Test	φ	-0.12	-0.19	0.09		**0.35**	0.10	-0.15	0.17	0.18
	p	0.45	0.24	0.59		**0.03***	0.55	0.35	0.30	0.26
Indirect Test	φ	0.20	-0.27	0.12	**0.35**		0.11	-0.07	**0.47**	0.26
	p	0.22	0.09	0.45	**0.03***		0.50	0.67	**0.00***	0.11
General Fixture	φ	0.12	**-0.36**	0.03	0.10	0.11		0.24	0.11	0.27
	p	0.50	**0.03***	0.86	0.55	0.50		0.13	0.50	0.10
Eager Test	φ	**0.37**	0.07	0.27	-0.15	-0.07	0.24		-0.07	0.17
	p	**0.02***	0.68	0.09	0.35	0.67	0.13		0.67	0.28
Lazy Test	φ	**0.47**	-0.03	**0.37**	0.17	**0.47**	0.11	-0.07		-0.09
	p	**0.00***	0.88	**0.02***	0.30	**0.00***	0.50	0.67		0.57
Duplicated Code	φ	0.26	**0.36**	-0.11	0.18	0.26	0.27	0.17	-0.09	
	p	0.11	**0.02***	0.48	0.26	0.11	0.10	0.28	0.57	

Appendix Table XV: Results of the φ correlation coefficient analysis between test smells in JMeter

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	ϕ		-0.08	0.19	-0.08	0.13	0.21	-0.24	0.14	0.13
	p		0.49	0.10	0.48	0.07	0.07	0.04*	0.24	0.26
Assertion Roulette	ϕ	-0.08		0.12	0.02	0.14	0.21	0.18	-0.06	0.12
	p	0.49		0.32	0.90	0.23	0.07	0.12	0.61	0.31
Sensitive Equality	ϕ	0.19	0.12		0.20	0.08	0.05	-0.01	0.04	0.10
	p	0.10	0.32		0.09	048	0.65	0.96	0.74	0.38
Mystery Test	ϕ	-0.08	0.02	0.20		0.06	-0.11	-0.13	-0.08	0.07
	p	0.48	0.90	0.09		0.64	0.32	0.25	0.51	0.56
Indirect Test	ϕ	0.13	0.14	0.08	0.06		-0.02	-0.00	**0.33**	0.25
	p	0.07	0.23	048	0.64		0.90	0.98	**0.00***	0.03*
General Fixture	ϕ	0.21	0.21	0.05	-0.11	-0.02		0.22	-0.03	0.25
	p	0.07	0.07	0.65	0.32	0.90		0.06	0.81	0.03*
Eager Test	ϕ	-0.24	0.18	-0.01	-0.13	-0.00	0.22		-0.10	0.07
	p	0.04*	0.12	0.96	0.25	0.98	0.06		0.39	0.57
Lazy Test	ϕ	0.14	-0.06	0.04	-0.08	**0.33**	-0.03	-0.10		0.02
	p	0.24	0.61	0.74	0.51	**0.00***	0.81	0.39		0.89
Duplicated Code	ϕ	0.13	0.12	0.10	0.07	0.25	0.25	0.07	0.02	
	p	0.26	0.31	0.38	0.56	0.03*	0.03*	0.57	0.89	

Appendix Table XVI: Results of the φ correlation coefficient analysis between test smells in JabRef

Smells		**Assertion-free**	**Assertion Roulette**	**Sensitive Equality**	**Mystery Test**	**Indirect Test**	**General Fixture**	**Eager Test**	**Lazy Test**	**Duplicated Code**
Assertion-free	φ	▨	-0.16	0.12	-0.15	-0.04	-0.14	**0.41**	-0.06	-0.08
	p	▨	0.25	0.37	0.27	0.75	0.30	**0.00***	0.65	0.58
Assertion Roulette	φ	-0.16	▨	0.09	-0.05	0.04	-0.07	-0.08	0.05	0.19
	p	0.25	▨	0.51	0.71	0.78	0.63	0.54	0.69	0.17
Sensitive Equality	φ	0.12	0.09	▨	0.02	-0.04	0.20	0.05	-0.06	**0.34**
	p	0.37	0.51	▨	0.91	0.75	0.13	0.72	0.65	**0.01***
Mystery Test	φ	-0.15	-0.05	0.02	▨	-0.06	**0.43**	**0.34**	0.16	-0.01
	p	0.27	0.71	0.91	▨	0.63	**0.00***	**0.01***	0.24	0.95
Indirect Test	φ	-0.04	0.04	-0.04	-0.06	▨	-0.06	-0.06	-0.03	-0.09
	p	0.75	0.78	0.75	0.63	▨	0.66	0.68	0.85	0.50
General Fixture	φ	-0.14	-0.07	0.20	**0.43**	-0.06	▨	-0.04	-0.09	-0.08
	p	0.30	0.63	0.13	**0.00***	0.66	▨	0.75	0.52	0.54
Eager Test	φ	**0.41**	-0.08	0.05	**0.34**	-0.06	-0.04	▨	0.20	0.17
	p	**0.00***	0.54	0.72	**0.01***	0.68	0.75	▨	0.15	0.21
Lazy Test	φ	-0.06	0.05	-0.06	0.16	-0.03	-0.09	0.20	▨	-0.13
	p	0.65	0.69	0.65	0.24	0.85	0.52	0.15	▨	0.34
Duplicated Code	φ	-0.08	0.19	**0.34**	-0.01	-0.09	-0.08	0.17	-0.13	▨
	p	0.58	0.17	**0.01***	0.95	0.50	0.54	0.21	0.34	▨

Appendix Table XVII: Results of the φ correlation coefficient analysis between test smells in Apache Commons Lang

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	φ		0.11	-0.09	-0.02	0.07	0.17	0.09	-0.02	0.00
	p		0.22	0.30	0.79	0.41	0.06	0.33	0.79	1.00
Assertion Roulette	φ	0.11		0.18	0.14	-0.05	0.00	0.11	-0.03	-0.07
	p	0.22		0.04	0.122	0.57	0.99	0.24	0.76	0.47
Sensitive Equality	φ	-0.09	0.18		-0.07	-0.12	0.03	0.03	-0.03	-0.07
	p	0.30	0.04		0.47	0.18	0.78	0.78	0.76	0.47
Mystery Test	φ	-0.02	0.14	0.05		-0.03	-0.08	0.04	-0.04	0.05
	p	0.79	0.122	0.55		0.77	0.38	0.63	0.64	0.58
Indirect Test	φ	0.07	-0.05	-0.12	-0.03		0.14	0.14	-0.03	0.20
	p	0.41	0.57	0.18	0.77		0.12	0.12	0.77	0.03*
General Fixture	φ	0.17	0.00	0.03	-0.08	0.14		0.07	-0.08	0.18
	p	0.06	0.99	0.78	0.38	0.12		0.46	0.38	0.05*
Eager Test	φ	0.09	0.11	0.03	0.04	0.14	0.07		0.29	0.28
	p	0.33	0.24	0.78	0.63	0.12	0.46		0.00*	0.00*
Lazy Test	φ	-0.02	-0.13	-0.03	-0.04	-0.03	-0.08	0.29		0.05
	p	0.79	0.16	0.76	0.64	0.77	0.38	0.00*		0.58
Duplicated Code	φ	0.00	0.20	-0.07	0.05	0.20	0.18	0.28	0.05	
	p	1.00	0.02*	0.47	0.58	0.03*	0.05*	0.00*	0.58	

Appendix Table XVIII: Results of the φ correlation coefficient analysis between test smells in Dependency Finder

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	φ		-0.21	0.02	0.06	-0.03	0.01	0.16	-------	-0.03
	p		0.00*	0.76	0.42	0.66	0.85	0.02*	-------	0.66
Assertion Roulette	φ	-0.21		-0.03	-0.05	0.18	0.18	0.15	-------	0.15
	p	0.00*		0.65	0.42	0.01*	0.01*	0.03*	-------	0.03
Sensitive Equality	φ	0.02	-0.03		-0.03	-0.08	0.02	0.09	-------	0.03
	p	0.76	0.65		0.68	0.26	0.81	0.18	-------	0.63
Mystery Test	φ	0.06	-0.05	-0.03		-0.09	-0.13	0.00	-------	0.13
	p	0.42	0.42	0.68		0.21	0.06	0.98	-------	0.05
Indirect Test	φ	-0.03	0.18	-0.08	-0.09		-0.01	0.00	-------	0.11
	p	0.66	0.01*	0.26	0.21		0.93	1.00	-------	0.11
General Fixture	φ	0.01	0.12	0.02	-0.13	-0.01		-0.03	-------	0.09
	p	0.85	0.08	0.81	0.06	0.93		0.62	-------	0.19
Eager Test	φ	0.16	0.15	0.09	0.00	0.00	-0.03		-------	-0.04
	p	0.02*	0.03*	0.18	0.98	1.00	0.62		-------	0.52
Lazy Test	φ	-------	-------	-------	-------	-------	-------	-------		-------
	p	-------	-------	-------	-------	-------	-------	-------		-------
Duplicated Code	φ	-0.03	0.15	0.03	0.13	0.11	0.09	-0.04	-------	
	p	0.66	0.03	0.63	0.05	0.11	0.19	0.52	-------	

Appendix Table XIX: Results of the φ correlation coefficient analysis between test smells in MOEA

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	φ		-0.04	-0.02	0.09	-0.03	-0.03	0.02	0.15	-0.09
	p		0.59	0.77	0.23	0.72	0.72	0.78	0.03*	0.20
Assertion Roulette	φ	-0.04		0.07	0.02	-0.01	-0.01	-0.06	0.01	0.17
	p	0.59		0.36	0.75	0.87	0.87	0.42	0.86	0.02*
Sensitive Equality	φ	-0.02	0.07		-0.04	-0.01	0.11	-0.03	-0.02	-0.07
	p	0.77	0.36		0.61	0.86	0.13	0.64	0.84	0.31
Mystery Test	φ	0.09	0.02	-0.04		0.21	0.22	0.04	-0.02	0.09
	p	0.23	0.75	0.61		0.00*	0.00*	0.55	0.80	0.22
Indirect Test	φ	-0.03	-0.01	-0.01	0.21		0.19	-0.04	-0.02	0.09
	p	0.72	0.87	0.86	0.00*		0.01*	0.57	0.80	0.22
General Fixture	φ	-0.01	0.19	0.11	0.22	0.19		0.07	0.15	0.12
	p	0.92	0.01*	0.13	0.00*	0.01*		0.33	0.03*	0.08
Eager Test	φ	0.02	-0.06	-0.03	0.04	-0.04	0.07		-0.05	0.17
	p	0.78	0.42	0.64	0.55	0.57	0.33		0.51	0.02*
Lazy Test	φ	0.15	0.01	-0.02	-0.05	-0.02	0.15	-0.05		-0.03
	p	0.03*	0.86	0.84	0.46	0.80	0.03*	0.51		0.72
Duplicated Code	φ	-0.09	0.17	-0.07	0.14	0.09	0.12	0.17	-0.03	
	p	0.20	0.02*	0.31	0.04*	0.22	0.08	0.02*	0.72	

Appendix Table XX: Results of the φ correlation coefficient analysis between test smells in Barcode4J

Smells		Assertion-free	Assertion Roulette	Sensitive Equality	Mystery Test	Indirect Test	General Fixture	Eager Test	Lazy Test	Duplicated Code
Assertion-free	ϕ		**-0.54**	0.18	-------	-0.10	-------	-0.06	-0.07	-0.21
	p		**0.00***	0.32	-------	0.57	-------	0.74	0.70	0.25
Assertion Roulette	ϕ	**-0.54**		0.18	-------	0.13	-------	0.34	0.09	0.27
	p	**0.00***		0.32	-------	0.47	-------	0.06	0.62	0.14
Sensitive Equality	ϕ	0.18	0.18		-------	-0.01	-------	0.25	0.18	0.21
	p	0.32	0.32		-------	0.96	-------	0.16	0.33	0.23
Mystery Test	ϕ	-------	-------	-------		-------	-------	-------	-------	-------
	p	-------	-------	-------		-------	-------	-------	-------	-------
Indirect Test	ϕ	-0.10	0.13	-0.01	-------		-------	0.10	-0.05	0.17
	p	0.57	0.47	0.96	-------		-------	0.58	0.79	0.34
General Fixture	ϕ	-------	-------	-------	-------	-------		-------	-------	-------
	p	-------	-------	-------	-------	-------		-------	-------	-------
Eager Test	ϕ	-0.06	0.34	0.25	-------	0.10	-------		0.27	0.12
	p	0.74	0.06	0.16	-------	0.58	-------		0.14	0.50
Lazy Test	ϕ	-0.07	0.09	0.18	-------	-0.05	-------	0.27		-0.10
	p	0.70	0.62	0.33	-------	0.79	-------	0.14		0.58
Duplicated Code	ϕ	-0.21	0.27	0.21	-------	0.17	-------	0.12	-0.10	
	p	0.25	0.14	0.23	-------	0.34	-------	0.50	0.58	

Appendix G: Size effect analysis using Bootstrapping

Appendix Table XXI: Results of the confounding effect of LOC on the relationship between the number of test smell types and TLOC

DV: Number of test smell types, IV: TLOC, M: LOC					
System	**Effect**	**SE(Boot)**	**LLCI**	**ULCI**	**Significant indirect effect (possibility of mediation)**
JFreeChart	0.001	0.000	0.001	0.002	Yes
FindBugs	0.000	0.001	-0.003	0.001	**No**
JMeter	0.000	0.000	-0.001	0.001	**No**
Commons Lang	0.000	0.000	0.000	0.000	**No**
JabRef	0.000	0.001	-0.003	0.002	**No**
Dependency Finder	0.000	0.000	0.000	0.000	**No**
MOEA	0.001	0.001	0.000	0.002	**No**
Barcode4J	0.000	0.004	-0.012	0.005	**No**

DV= Dependent Variable   IV: Independent Variable

Appendix Table XXII: Results of the confounding effect of LOC on the relationship between TLOC and the number of code smell types

DV: TLOC IV: Number of code smell types, M: LOC					
System	**Effect**	**SE(Boot)**	**LLCI**	**ULCI**	**Significant indirect effect (possibility of mediation)**
JFreeChart	22.334	10.545	2.144	43.846	Yes
FindBugs	-6.156	8.693	-17.795	16.267	**No**
JMeter	22.162	26.264	-14.289	88.513	**No**
Commons Lang	99.104	57.205	9.070	237.513	Yes
JabRef	32.069	58.349	-70.273	161.197	**No**
Dependency Finder	29.986	19.866	-0.360	75.814	**No**
MOEA	14.532	6.747	2.781	29.688	Yes
Barcode4J	20.001	12.170	-14.658	43.749	**No**

Appendix Table XXIII: Results of the confounding effect of TLOC on the relationship between the number of test smell types and LOC

DV: Number of test smell types, IV: LOC, M: TLOC					
System	**Effect**	**SE(Boot)**	**LLCI**	**ULCI**	**Significant indirect effect (possibility of mediation)**
JFreeChart	0.000	0.000	-0.001	0.000	**No**
FindBugs	0.000	0.002	-0.002	0.004	**No**
JMeter	0.002	0.002	0.000	0.007	**No**
Commons Lang	0.000	0.001	-0.001	0.002	**No**
JabRef	0.003	0.003	-0.003	0.009	**No**
Dependency Finder	0.000	0.000	0.000	0.001	**No**
MOEA	0.001	0.001	0.000	0.004	**No**
Barcode4J	0.008	0.005	0.001	0.019	Yes

Appendix Table XXIV: Results of the confounding effect of LOC on the relationship between the number of test smell types and CC

DV: Number of test smell types, IV: CC, M: LOC					
System	**Effect**	**SE(Boot)**	**LLCI**	**ULCI**	**Significant indirect effect (possibility of mediation)**
JFreeChart	0.033	0.026	0.006	0.093	Yes
FindBugs	-0.023	0.061	-0.176	0.091	**No**
JMeter	0.010	0.028	-0.025	0.107	**No**
Commons Lang	0.063	0.030	0.023	0.151	Yes
JabRef	0.006	0.013	-0.005	0.036	**No**
Dependency Finder	0.014	0.011	-0.002	0.043	**No**
MOEA	0.046	0.023	0.014	0.093	Yes
Barcode4J	0.057	0.082	-0.161	0.335	**No**

Appendix Table XXV: Results of the confounding effect of LOC on the relationship between the number of test smell types and the number of code smell types

DV: Number of test smell types, IV: Number of code smell, M: LOC					
System	**Effect**	**SE(Boot)**	**LLCI**	**ULCI**	**Significant indirect effect (possibility of mediation)**
JFreeChart	0.117	0.034	0.056	0.188	Yes
FindBugs	-0.243	0.120	-0.458	0.032	**No**
JMeter	0.025	0.114	-0.191	0.250	**No**
Commons Lang	-0.182	0.089	-0.351	0.003	**No**
JabRef	0.215	0.298	-0.321	0.821	**No**
Dependency Finder	0.039	0.053	-0.067	0.136	**No**
MOEA	0.026	0.067	-0.091	0.174	**No**
Barcode4J	0.018	0.346	-0.575	0.765	**No**