A Study of the Relationship Between Class Testability and Runtime Properties

Conference Paper · December 2015

DOI: 10.1007/978-3-319-27218-4_5

3 authors:

Amjed Tahir
Massey University
26 PUBLICATIONS **109** CITATIONS
SEE PROFILE

Stephen G. MacDonell
Auckland University of Technology/University of Otago
253 PUBLICATIONS **2,858** CITATIONS
SEE PROFILE

Jim Buchan
Auckland University of Technology
25 PUBLICATIONS **108** CITATIONS
SEE PROFILE

A Study of the Relationship Between Class Testability and Runtime Properties

Amjed Tahir[1](✉), Stephen MacDonell[1,2], and Jim Buchan[2]

[1] Department of Information Science, University of Otago, Dunedin, New Zealand
amjed.tahir@otago.ac.nz

[2] SERL, School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand
{smacdone,jbuchan}@aut.ac.nz

Abstract. Software testing is known to be expensive, time consuming and challenging. Although previous research has investigated relationships between several software properties and software testability the focus has been on static software properties. In this work we present the results of an empirical investigation into the possible relationship between runtime properties (dynamic coupling and key classes) and class testability. We measure both properties using dynamic metrics and argue that data gathered using dynamic metrics are both broader and more precise than data gathered using static metrics. Based on statistical analysis, we find that dynamic coupling and key classes are significantly correlated with class testability. We therefore suggest that these properties could be used as useful indicators of class testability.

Keywords: Testability · Unit testing · Dynamic metrics · Dynamic coupling · Program comprehension

1 Introduction

Software testing activities typically require significant time and effort in both planning and execution. Testing is thus acknowledged to be expensive [1] as it can consume up to 50 % of the total cost and effort in a software development project [2]. Although software systems have been growing larger and more complex [3], testing resources, by comparison, have remained limited and constrained [4]. Software components with low-level testability may be less trustworthy, even after successful testing [1]. Understanding and reducing testing effort have therefore been enduring fundamental goals for both academic and industrial research.

The notion that a software product has properties that are related to the effort needed to validate that product is commonly referred to as the ‘testability’ of that product [5]. Binder [6] coined the phrase “Design for Testability” to describe software construction that considers testability from the early stages of development. The core expectation is that software components with a high degree of testability are easier to test and consequently will be more effectively tested, raising the software quality compared to software that has lower testability. Improving software testability should help to reduce testing cost, effort, and demand for resources. If components are difficult to test, then the size of the test cases designed to test those components, and the required testing effort, will necessarily be larger [7]. Components with poor testability are also more expensive to repair when problems are detected late in the development process. In contrast, components and software with good testability can dramatically increase the quality of the software as well as reduce the cost of testing [8].

While clearly a desirable trait, testability has been recognized as being an elusive concept, and its measurement and evaluation are acknowledged as being challenging endeavors [4]. Researchers have therefore identified numerous factors that (may) have an impact on the testability of software. For instance, software testability is claimed to be affected by the extent of the required validation, the process and tools used, and the representation of the requirements, among other factors [9]. Given their diversity it is challenging to form a complete and consistent view on all the potential factors that may affect testability as well as the degree to which these factors are present and influential under different testing contexts. Several are considered here to illustrate the breadth of factors that potentially influence testability.

A substantial body of work has addressed a diversity of design and code characteristics that can affect the testability of a software product. For example, the relationships been internal class properties in Object Oriented (OO) systems and characteristics of the corresponding unit tests have been investigated in several previous studies (e.g., [9, 10]). In these studies, several OO design metrics (drawn mainly from the C&K suite [11]) have been used to investigate 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 [9] in such research.

In their research, Bruntink and van Deursen [9] used only static software measures, and this is the case for all previous work in this area. Others, such as Basili et al. [12], take the view that traditional static software metrics may be necessary but not sufficient for characterizing, assessing and predicting the quality profile of OO systems. In this paper we build on that view and propose the use of dynamic metrics to represent further quality characteristics. Dynamic metrics are the sub-class of software measures that capture the dynamic behavior of a software system and have been shown to be related to software quality attributes [13, 14]. Consideration of this group of metrics provides a more complete insight into the multiple dimensions of software quality when compared to static metrics alone [15]. Dynamic metrics are usually computed based on data collected during program execution (i.e. at runtime). Therefore they can directly reflect the quality attributes of that program, product or system in operation. This paper extends the investigation of software characteristics as factors in code testability by characterizing the code using dynamic metrics. A fuller discussion of dynamic metrics and their relative advantages over static metrics is presented in [16].

The work presented here extends our previous work [17] by adding one more system to the original three systems analyzed [17]. We also improve our analysis and discussion by using a different statistical test of correlation and by providing additional graphical representations of the data.

The rest of the paper is structured as follows. Section 2 provides the research context for this paper by reviewing related work, and confirms the potential of relating dynamic code metrics to testability. Section 3 argues for the suitability of the *dynamic coupling* and *key classes* concepts as appropriate dynamic metrics to characterize the code in relation to testability. These metrics are then used in the design of a set of experiments to test our hypotheses on specific case systems, as described in Sects. 4 and 5. The results of these experiments are then presented in Sect. 6 and their implications are discussed in Sect. 7. Threats to the study's validity are noted in Sect. 8. Finally, the main conclusions from the study and some thoughts on related future work are presented in Sect. 9.

2 Related Work

Several previous works have investigated the relationships between properties of software production code components and properties of their associated test code, with an emphasis on unit tests. The focus of that work has varied from designing measures for testability and testing effort to assessing the strength of the relationships between them. Given constraints on space, we consider a few typical studies here. Our intent is to be illustrative as opposed to exhaustive, and these studies are representative of the larger body of work in this research domain.

Bruntink and van Deursen [9] investigated the relationship between several OO metrics and class testability for the purpose of planning and estimating later testing activities. The authors found a strong correlation between class-level metrics, such as Number of Methods (NOM), and test level metrics, including the number of test cases and the lines of code per test class. Five different software systems, including one open source system, were traversed during their experiments. However, no evidence of relationships was found between inheritance-related metrics, e.g., Coupling Between Objects (CBO), and the proposed testability metrics. This is likely to be because the test metrics were considered at the class level. These inheritance-related metrics are expected to have a strong correlation with testability *at the integration and/or system level*, as polymorphism and dynamic binding increase the complexity of a system and the number of required test cases, and contribute to a consequent decrease in testability [4]. This suggestion can only be confirmed through evaluation at the object level using dynamic metrics. In a similar study, Badri et al. [10] investigated the relationship between cohesion and testability using the C&K static Lack of Cohesion metric. They found a significant positive correlation between static cohesion and software testability, where testability was measured using the metrics suggested by [9]. More recently, Zhou et al. [18] found that most structural code metrics that are obtained from static analysis are statistically correlated with class testability, with class size being the strongest indicator of class testability.

In other work related to testability, Arisholm et al. [19] found significant relationships between dynamic coupling measures, especially Dynamic Export Coupling, and change-proneness. Export Coupling appears to be a significant indicator of change-proneness and likely complements existing coupling measures based on static analysis (i.e., when used with size and static coupling measures).

3 Testability Concepts

In this work we investigate two runtime properties that we contend are related to class testability, and in the following we describe these and justify their suitability.

3.1 Dynamic Coupling

In this study dynamic coupling has been selected as one of the system characteristics to measure and investigate regarding its relationship to testability. Coupling has been shown in prior work to have a direct impact on the quality of software, and is also related to the software quality characteristics of complexity and maintainability [20, 21]. 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 [22, 23]. This suggests that it is reasonable to expect that coupling will be related to testability. Dynamic rather than static coupling has been selected for our investigation to address some shortcomings of the traditional static measures of coupling. For many years coupling has been measured statically, based on the limited structural properties of software [24]. This misses the coupling at runtime between different components at different levels (classes, objects, packages, and so on), which should capture a more complete picture and so relate better to testability. This notion of measuring dynamic coupling is quite common in the emergent software engineering research literature. In our recent systematic mapping study of dynamic metrics, dynamic coupling was found to be the most widely investigated system characteristic used as a basis for dynamic analysis [16].

For the purposes of this work the approach taken by [19] is followed, and dynamic coupling metrics that capture coupling at the object level are used. Two objects are coupled if at least one of them acts upon the other [11]. The 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*), or vice versa. Details of the specific metrics used to measure this form of coupling are provided in Sect. 4.2.

3.2 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 systems are formed around groups of classes some of which are linked together. As software systems grow in size, so the number of classes used increases in these systems. To analyze and understand a program or a system, how it works and the potential for decay, it is important to know where to start and which aspects should be given priority. From a maintenance perspective, understanding the roles of classes and their relative importance to a system is essential. In this respect there are classes that could have more influence and play more prominent roles than others. This group of classes is referred to here 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 the more effective planning of testing activities. One of the potential usages of these classes is in prioritizing testing activities – testers could usefully prioritize their work by focusing on testing these Key Classes first, alongside consideration of other factors such as risk and criticality information.

The concept of Key Classes is seen elsewhere in the literature but has an important distinction in meaning and usage in this research. For example, in [24], classification as a Key Class is based on the level of coupling of a class. Therefore, Key Classes are those classes that are tightly coupled. In contrast, our definition is based on the *usage* of these classes: Key Classes are those classes that have high execution frequency at runtime. A metric used to measure Key Classes is explained in Sect. 4.2.

4 Study Design

In this section we explain our research questions and the hypotheses that the work is aimed at testing. We also define the various metrics used in operational terms and our analysis procedures.

One of the key challenges faced when evaluating software products is the choice of appropriate measurements. Metric selection in this research has been determined in a goal-oriented manner using the GQM framework [25] and its extension, the GQM/MEDEA framework [26]. Our ***goal*** is to better understand what affects software testability, and our ***objective*** is to assess the presence and strength of the relationship between dynamic coupling and key classes on the one hand and code 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 open source systems.

4.1 Research Questions and Hypotheses

We investigate two factors that we contend are, in principle, related to system testability: dynamic coupling and key classes. For this purpose, we have two research questions to answer:

RQ1: Is *dynamic coupling* of a class significantly correlated with the class testability of its corresponding test class/unit?

RQ2: Are *key classes* significantly correlated with the class testability of their corresponding test classes/units?

The following two research hypotheses are investigated to answer the research questions:

H0: Dynamic coupling has a significant correlation with class testability.

H1: Key classes have a significant correlation with class testability.

The corresponding null hypotheses are:

H2: Dynamic coupling has no significant correlation with class testability measures.
H3: Key Classes have no significant correlation with class testability.

4.2 Measurements

In Sect. 3 we described the *dynamic coupling* and *key classes* concepts. In this section we define specific dynamic metrics that can be used to measure these concepts. We also explain the metrics used to measure class testability.

Dynamic Coupling Metrics. As stated in Sect. 3.1, dynamic coupling is intended to be measured in two forms - when a class is accessed by another class at runtime, and when a class accesses other classes at runtime (i.e., to account for both *callers* and *callees*). To measure these levels of coupling we select the previously defined *Import Coupling (IC)* and *Export Coupling (EC)* metrics [19]. IC measures the number of method invocations ***received*** by a class (*callee*) from other classes (*callers*) in the system. EC measures the number of method invocations ***sent*** from a class (*caller*) to other classes (*callees*) in the system. Note that both metrics are collected based on method invocations/calls. More detailed explanations of these metrics are provided in [19].

Key Classes Metrics. The concept of Key Classes is explained in Sect. 3.2. The goal here is to examine if those Key Classes (i.e., those classes with higher frequency of execution) have a significant relationship with class testability (as defined in the next subsection). We define the *Execution Frequency (EF)* dynamic metric to identify those Key Classes. EF for class C counts the number of executions of methods within class C. Consider a class C, with methods $m1$, $m2$,….. mn. Let EF(mi) be the number of executions of method m of class C, then:

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

where n is the number of executed methods within class C.

Class Testability Measures. The testability of a class is considered here in relation to unit tests. In this work, we utilize two static metrics to measure unit test characteristics: Test Lines of Code (TLOC) and the Number of Test Cases (NTC). These metrics are motivated by the test suite metrics suggested by [9]. TLOC, derived from the classic Lines of Code (LOC) metric, is a size measure that counts the total number of physical lines of code within a test class or classes. NTC is a test design metric that counts the total number of test cases in a test class. Our hypotheses thus reflect an expectation that the dynamic coupling and key classes of production code classes are related to the size and scope of their associated test classes.

Our data collection methods are explained in more detail in the following section.

5 Data Collection

The collection of dynamic metrics data can be accomplished in various ways. The most common (and 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 Aspect Oriented Programming (AOP). Previous works (including [23, 27, 28]) 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.

Testability metrics data, including LOC, TLOC, and Number of Classes (NOC), are collected using the *CodePro Analytix*[1] tool and the values were later checked and verified using the *Eclipse Metrics Plugin*[2]. Values for the NTC metric are collected from the *JUnit* framework and these values were verified manually by the first author.

We used the two different traceability techniques suggested by [29] to identify unit test classes and link them to their corresponding production classes. First, we used the *Naming Convention* technique to link test classes to production classes following their names. It has been widely suggested (for instance, in the JUnit documentation) that a test class should be named after the corresponding class(es) that it tests, by adding "Test" to the original class name. Second, we used a *Static Call Graph* technique, which inspects method invocations in the test case. The latter process was carried out manually by the first author. The effectiveness of the Naming Convention technique is reliant on developers' efforts in conforming to a coding standard, whereas the Static Call Graph approach reveals direct references to production classes in the test classes.

It is important to note here that we only consider core system code: only production classes that are developed as a part of the system are assessed. Additional classes (including those in jar files) 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.

5.1 Case Studies

To consider the potential relationships between class testability and the chosen dynamic metrics we selected four different open source systems to be used in our experiments. Selection of these systems was conducted with the goal of examining applications of reasonable size, with some degree of complexity, and easily accessible source code. The main criteria for selecting the applications are: (1) each application should be fully open source, i.e., source code for both production code and test code is publicly available; (2) each application must be written in Java, as we are using the JUnit and AspectJ frameworks, which are both written for Java; (3) each application

[1] https://developers.google.com/java-devtools/codepro/doc/

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

should come with test suites; and (4) each application should comprise at least 25 test classes.

The systems selected for our experiments are: *FindBugs, JabRef, Dependency Finder and MOEA.* Table 1 reports particular characteristics and size information of both the production and test code of the four systems.

Table 1. Characteristics of the selected systems.

System	Version	KLOC	Size	NOC	# Unit Tests	Test KLOC
FindBugs	2.0.3	117	Large	1245	46	2.683
JabRef	2.9.2	90.4	Medium	616	55	5.392
Dependency finder	1.2.1 beta4	58	Medium	450	258	32.095
MOEA	1.17	42	Medium	407	280	16.694

The size classification used in Table 1 is adapted from the work of [30], where application size is categorized into bands based on the number of kilo LOC (KLOC): small (fewer than 1 KLOC), medium (1–10 KLOC), large (10–100 KLOC) and extra-large (more than 100 KLOC).

5.2 Execution Scenarios

In order to arrive at dynamic metrics values that are associated with typical, genuine use of a system the selected execution scenarios must be representative of such use. Our goal is to mimic 'actual' system behavior, as this will enhance the utility of our results. The scenarios are therefore designed to use key system features, based on the available documentation and user manuals for the selected systems, as well as our prior knowledge of these systems. Further information on the selected execution scenario for each system now follows. Note that all four systems have GUI access, and the developed scenarios assume use via the GUI.

FindBugs: The tool is run to detect bugs in a large scale OSS (i.e., JFreeChart) by analyzing the source code and the associated jar files. The web plugin has been 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. We included all reference types supported by the tool (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 added and used during this execution.

Dependency Finder: this scenario involves using the tool to analyze the source code of four medium-large sized systems one after another, namely, FindBugs, JMeter, Ant and Colossus. We computed dependencies (dependency graphs) and OO metrics at all layers (i.e., packages, classes, features). 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. We used this diagnostic tool to apply those different algorithms on the predefined problems. We applied each of these algorithms at least once on each problem. We displayed metrics and performance indicators for all results provided by those different problems and algorithms. Statistical results of these multiple runs were displayed at the end of the analysis.

6 Results

As we are interested in the potential associations between variables, a statistical test of correlation is used in the evaluation of our hypotheses. After collecting our metrics data we first apply the Shapiro-Wilk (S-W) test to check the normality of each data distribution. This is necessary as selection of the relevant correlation test should be informed by the nature of the distributions, being normal or non-normal. The null hypothesis for the S-W test is that data is normally distributed.

After applying the S-W test the evidence led us to reject the null hypothesis regarding their distribution, and so we accepted that the data were not normally distributed (boxplots of the data are shown in Figs. 1 and 2). We therefore decided to use *Spearman's rho (r)* rank correlation coefficient test. *Spearman's r* is a non-parametric statistical test that measures the association between two measured quantities when ordered and ranked. In our work *Spearman's r* 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 Sect. 4.2.

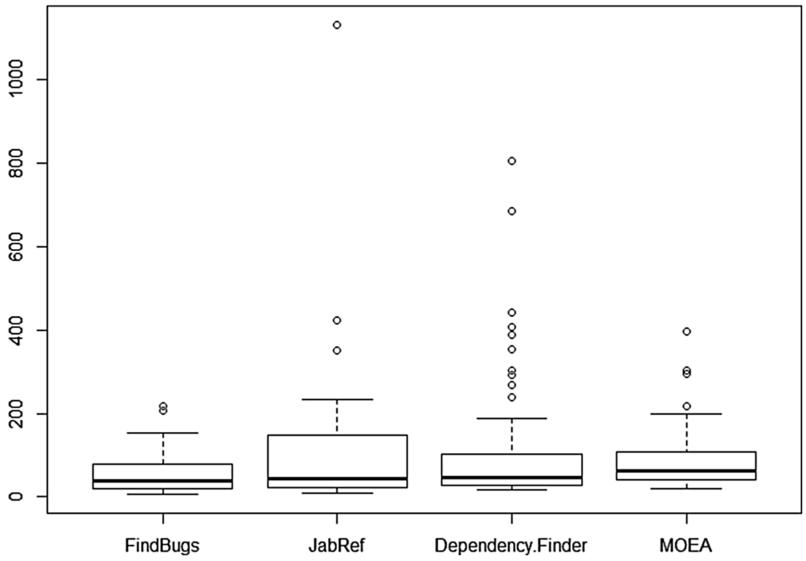

Fig. 1. Boxplots of TLOC in all four systems.

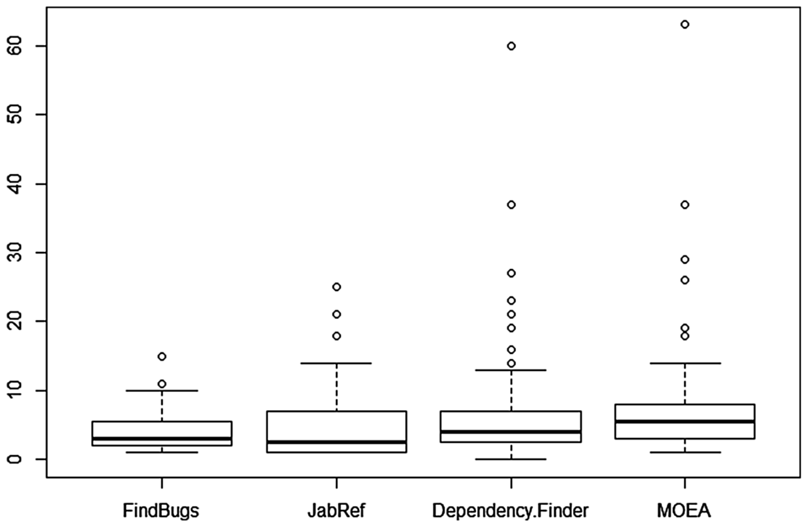

Fig. 2. Boxplots of NTC in all four systems.

We used the classification of Cohen [31] to interpret the degree of association between variables. The value of r indicates the association between two ranked variables, and it ranges from −1 (perfect negative correlation) to +1 (perfect positive correlation). We interpret that variables are independent when $r = 0$, that there is a low direct association when $r < 0.3$, a medium direct association when $0.3 \leq r < 0.5$, and a high direct association when $0.5 \leq r$. This interpretation also applies to negative correlations, but the association is inverse rather than direct [32]. The p-value (p) represents the statistical significance of the relationship. We consider an association to be statistically significant where $p \leq 0.05$.

The number of observations considered in each test varies in accordance with the systems' execution scenarios described in Sect. 5.2. Observation points, in fact, represent the number of 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). The number of observations for *FindBugs* is 23, *JabRef* is 26, 80 for *Dependency Finder* and 76 for *MOEA*.

Table 2 shows the *Spearman's r* results for the two dynamic coupling metrics against the test suite metrics. Corresponding results for the EF metric against the test suite metrics are presented in Table 3. For all analyses we interpret 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.

For dynamic coupling, a mix of results is found from the collected metrics (Table 2). 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* direct 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 are *medium* associations).

Table 2. *Spearman r* correlation between dynamic coupling metrics and class testability metrics.

Systems	Metrics	TLOC		NTC	
		r	*p*	*r*	*p*
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

In terms of relationships with the IC metric (Table 2), the 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 3. *Spearman r* correlation between EF metrics and class testability metrics.

Systems	Metrics	TLOC		NTC	
		r	*p*	*r*	*p*
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

As shown in Table 3, 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*, where a *low* correlation is found in *Dependency Finder*.

7 Discussion

Based on our analysis $\boldsymbol{H_0}$ is accepted and $\boldsymbol{H_2}$ 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 be significantly associated with the testability metrics for three of the four systems considered, $\boldsymbol{H_1}$ is also accepted and $\boldsymbol{H_3}$ is rejected on the balance of evidence. The relationships between coupling and class-testability metrics are shown in Figs. 3 and 4. Due to space constraints we show only *Scatter Plot* graphs from systems that have the *highest* correlations (i.e., *r* values) between metrics. Figure 3 shows the relationship between EC and NTC in *FindBugs* and Fig. 4 shows the relationship between EC and TLOC in *Dependency Finder*.

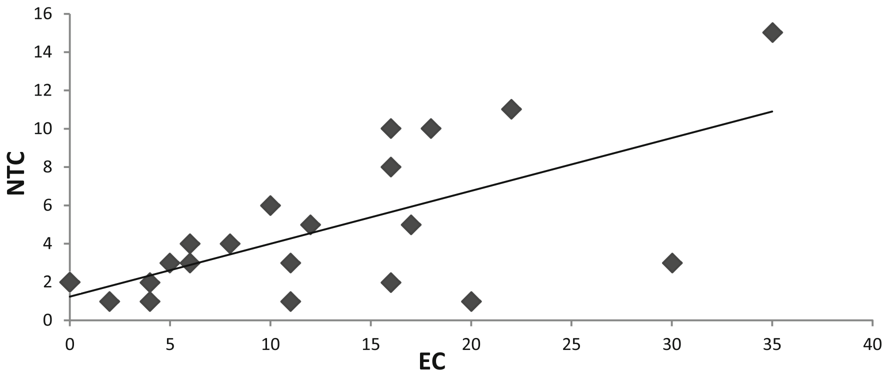

Fig. 3. Scatter plot of the relationship between EC and NTC metrics in *FindBugs*.

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 needs to 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 4) to varying degrees for the four systems investigated. 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*).

It is evident that dynamic coupling measures are associated with class-testability metrics. EC is found to be more significantly correlation with both testability metrics. IC association with class testability metrics is not consistence across systems. These results can be interpreted as indicating that dynamic coupling, in some form, has a significant correlation with class testability. A similar inference is drawn regarding key classes; this property is also significantly associated with class testability or unit test size. Additionally, the two dynamic testability concepts studied here, i.e., dynamic coupling and key classes, are found to be themselves significantly correlated. Such results can be helpful for testers and maintainers as they provide empirical evidence regarding the relationship between two important dynamic properties and class

Table 4. *Spearman r* results for the correlation between coupling and EF.

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

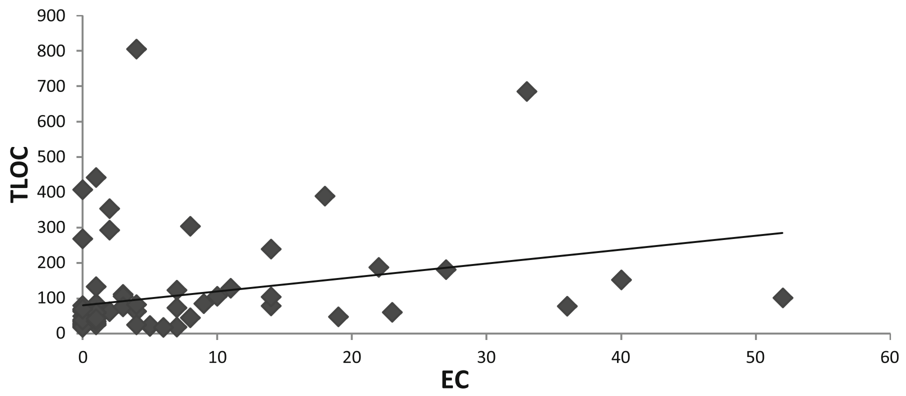

Fig. 4. Scatter plot of the relationship between EC and TLOC metrics in *Dependency Finder*.

testability. We recommend that similar dynamic information should be taken into consideration when developing unit tests or maintaining existing test suites.

In revisiting the list of the investigated research questions, dynamic coupling is found to have a significant (although not very strong) direct association with testability (RQ1). A more significant correlation was found between key classes (i.e., frequently executed classes) and class testability metrics. By answering RQ1 and RQ2, this suggests that dynamic coupling and key classes can act, to some extent, as complementary indicators of class testability (i.e., unit test size). It is contended here that a tightly coupled or frequently executed class would need a large corresponding test class (i.e., higher numbers of TLOC and NTC). This expectation has been found to be evidenced in at least three of the four systems examined.

8 Threats to Validity

We acknowledge the following threats that could affect the validity of our results.

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 systems, could enable further evaluation of the associations revealed in this study and so lead to more generalizable conclusions.

Unit test selections can be another validity threat. We only considered production classes that have corresponding test classes, which may lead to a selection bias. Classes that are extremely difficult to test, or are considered too simple, might have no associated test classes. Such production classes are not considered in our analyses. Due to their availability, we only included classes that had associated JUnit test classes, and ignored all others.

The selection of the execution scenarios is another possible threat to the validity of our results. We designed execution scenarios that mimic as closely as possible ‘actual’ system behavior, based on the available system documentation and, in particular, indications of each system’s key features. We acknowledge, however, that the selected scenarios might not be fully representative of the typical uses of the systems. Analyzing 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 our analyses.

Finally, we acknowledge that only available test information from the selected systems was used. We did not collect or have access to any information regarding the testing strategy of the four systems. Test strategy and criteria information could be very useful 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 implemented test strategy.

9 Conclusions and Future Work

In this work we set out to investigate the presence and significance of any associations between two runtime code properties, namely Dynamic Coupling and Key Classes, and the testability of classes in four open source OO systems. Testability was measured based on the systems' production classes and their associated unit tests. Two different metrics were used to measure class testability, namely TLOC and NTC. As we were interested in the relationships between system characteristics at runtime, dynamic coupling and key classes were measured using dynamic software metrics collected via AOP. Results were then analyzed statistically using the *Spearman's r* correlation coefficient test to study the associations.

The resulting evidence indicates that there is a significant association between dynamic coupling and internal class testability. We found that dynamic coupling metrics and especially the Export Coupling metric have a significant direct association with TLOC. A less significant association was found between dynamic Import Coupling and NTC. Similarly, Key Classes are also shown to be significantly associated with our test suite metrics in at least three of the four systems examined.

The findings of this work contribute to our understanding of the nature of the relationships between characteristics of production and test code. The use of dynamic measures can provide a level of insight that is not available using static metrics alone. These relationships can act as an indicator for internal class level testability, and should be of help in informing maintenance and reengineering tasks.

Several future directions are suggested by the outcomes of this research. This work can be extended by examining a wider range of systems (such as closed-source systems) to enable further evaluation of the findings. Another research direction would be to investigate whether *dynamic coupling* and *key classes* information can be used together to predict the size and structure of test classes. Predicting class-level testability should improve the early estimation and assessment of the effort needed in testing activities. 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 the current information about class testability with other testing information such as test coverage and test strategy.

References

1. Bertolino, A., Strigini, L.: On the use of testability measures for dependability assessment. IEEE Trans. Softw. Eng. **22**(2), 97–108 (1996)
2. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, p. 240. Wiley Publishing, New York (2011)
3. Sommerville, I., et al.: Large-scale complex IT systems. Commun. ACM **55**(7), 71–77 (2012)
4. Mouchawrab, S., Briand, L.C., Labiche, Y.: A measurement framework for object-oriented software testability. Inf. Softw. Technol. **47**(15), 979–997 (2005)
5. ISO, Software engineering - Product quality-Part 1. In: Quality model 2001, International Organization for Standardization Geneva
6. Binder, R.V.: Design for testability in object-oriented systems. Commun. ACM **37**(9), 87–101 (1994)
7. Traon, Y.L., Robach, C.: From hardware to software testability. In: International Test Conference on Driving Down the Cost of Test, pp. 710–719. IEEE Computer Society (1995)
8. Gao, J.Z., Jacob, H.-S., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House Publishers, Norwood (2003)
9. Bruntink, M., van Deursen, A.: An empirical study into class testability. J. Syst. Softw. **79** (9), 1219–1232 (2006)
10. Badri, L., Badri, M., Toure, F.: An empirical analysis of lack of cohesion metrics for predicting testability of classes. Int. J. Softw. Eng. Appl. **5**(2), 69–86 (2011)
11. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
12. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
13. Cai, Y.: Assessing the effectiveness of software modularization techniques through the dynamics of software evolution. In: 3rd Workshop on Assessment of COntemporary Modularization Techniques, Orlando (2008)
14. Scotto, M., et al.: A non-invasive approach to product metrics collection. J. Syst. Architect. **52**(11), 668–675 (2006)
15. Dufour, B., et al.: Dynamic metrics for java. In: 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 149–168. ACM, Anaheim (2003)
16. Tahir, A., MacDonell, S.G.: A systematic mapping study on dynamic metrics and software quality. In: International Conference on Software Maintenance. IEEE Computer Society (2012)
17. Tahir, A., MacDonell, S.G., Buchan, J.: Understanding class-level testability through dynamic analysis. In: 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 38–47, Lisbon (2014)
18. Zhou, Y., et al.: An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. Sci. China Inf. Sci. **55**(12), 2800–2815 (2012)
19. Arisholm, E., Briand, L.C., Foyen, A.: Dynamic coupling measurement for object-oriented software. IEEE Trans. Softw. Eng. **30**(8), 491–506 (2004)
20. Offutt, J., Abdurazik, A., Schach, S.: Quantitatively measuring object-oriented couplings. Softw. Qual. J. **16**(4), 489–512 (2008)
21. Al Dallal, J.: Object-oriented class maintainability prediction using internal quality attributes. Inf. Softw. Technol. **55**(11), 2028–2048 (2013)

22. Chaumun, M.A., et al.: Design properties and object-oriented software changeability. In: European Conference on Software Maintenance and Reengineering, p. 45. IEEE Computer Society (2000)
23. Tahir, A., Ahmad, R., Kasirun, Z.M.: Maintainability dynamic metrics data collection based on aspect-oriented technology. Malays. J. Comput. Sci. **23**(3), 177–194 (2010)
24. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using webmining techniques. J. Softw. Maintenance Evol. **20**(6), 387–417 (2008)
25. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. IEEE Trans. Softw. Eng. **10**(6), 728–738 (1984)
26. Briand, L.C., Morasca, S., Basili, V.R.: An operational process for goal-driven definition of measures. IEEE Trans. Softw. Eng. **28**(12), 1106–1125 (2002)
27. Cazzola, W., Marchetto, A.: AOP-HiddenMetrics: separation, extensibility and adaptability in SW measurement. J. Object Technol. **7**(2), 53–68 (2008)
28. Adams, B., et al.: Using aspect orientation in legacy environments for reverse engineering using dynamic analysis–an industrial experience report. J. Syst. Softw. **82**(4), 668–684 (2009)
29. Rompaey, B.V., Demeyer S.: Establishing traceability links between unit test cases and units under test. In: European Conference on Software Maintenance and Reengineering, pp. 209–218. IEEE Computer Society, Kaiserslautern (2009)
30. Zhao, L., Elbaum, S.: A survey on quality related activities in open source. SIGSOFT Softw. Eng. Notes **25**(3), 54–57 (2000)
31. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences, 2nd edn. Lawrence Erlbaum Associates, London (1988)
32. Daniel, W.W.: Applied Nonparametric Statistics. KENT Publishing Company, Boston (2000)