HOW DEVELOPERS TEST THEIR OPEN SOURCE SOFTWARE PRODUCTS
A Survey of Well-known OSS Projects

Davide Tosi
University of Insubria, Department of Informatica e Comunicazione, Varese, Italy

Abbas Tahir
Siemens AG, Munich, Germany

Keywords: Open source software, Testing, Survey, Testing frameworks, Aspect-oriented programming, Dynamic measures.

Abstract: Open Source Software (OSS) projects do not usually follow the traditional software engineering development paradigms found in textbooks, thus influencing the way OSS developers test their products.
In this paper, we explore a set of 33 well-known OSS projects to identify how software quality assurance is performed under the OSS model. The survey investigates the main characteristics of the projects and common testing issues to understand whether a correlation exists between the complexity of the project and the quality of its testing activity. We compare the results obtained in our survey with the data collected in a previous survey by L. Zhao and S. Elbaum. Our results confirm that OSS is usually not validated enough and therefore its quality is not revealed enough.
To reverse this negative trend, the paper suggests the use of a testing framework that can support most of the phases of a well-planned testing activity, and describes the use of Aspect Oriented Programming (AOP) to expose dynamic quality attributes of OSS projects.

1 INTRODUCTION

Software quality is becoming nowadays one of the main differentiation factors between similar software products. As an activity of the software quality assurance process, a software product is normally validated against its specified requirements. Software testing can obviously help during verification and validation.

IEEE Standard for Software Test Documentation (IEEE, 1998) defines software testing as: ”The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item”. The main goal of testing is to detect software failures. Testing activities are performed to ensure that software design, code, and documentation meet all the requirements imposed on them (Lewis, 2004). Software testing plays an important role in providing evidence on the degree to which the requirements have been met. Testing can only prove that software does not function properly under specific conditions but it can not prove that it functions properly under all conditions. Open Source Software (OSS), compared to closed source software (CSS), is different with regard to development. These differences restrict the applicability of the well established testing approaches developed for CSS in the open source domain.

Zhao and Elbaum surveyed quality related activities in open source (Zhao and Elbaum, 2000), and they reported that ”more than 80% of the OSS developer responded that their products do not have testing plans. However most of the products spend close to 40% of their lifetime in the testing stage. Only 21% of the products spend less than 20% of their lifetime in testing.” These findings imply that most of the OSS projects spend plenty of time performing testing activities. The question with this regard is: How systematically do they address testing?

In this paper we investigate 33 well-established

Tosi D. and Tahir A. (2010).
HOW DEVELOPERS TEST THEIR OPEN SOURCE SOFTWARE PRODUCTS - A Survey of Well-known OSS Projects.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 22-31
DOI: 10.5220/0002919600220031

OSS projects to explore how they address software testing. The investigation relies basically on the information available in the projects repositories.We did this analysis in the context of the european research project QualiPSo [http://www.qualipso.org/] . Our results confirmed the results of Zhao and Elbaum in which OSS is still usually not systematically tested. To reverse this negative trend, we suggest the utilization of comprehensive testing frameworks that support testing at different levels. Such frameworks provide an integrated, single-entry environment for most of the testing activities. In this paper, we investigate the well-known and frequently used test framework, the Eclipse Test and Performance Testing Tools (TPTP). TPTP is an open source test tools platform that covers nearly all of the testing levels for software implemented in Java.

In particular, the TPTP platform supports dynamic testing. The main purpose of dynamic testing is to provide confidence on the dynamic behavior of the software. Therefore testing provides a way to evaluate some dynamic attributes of the software under test. Another way to have insight into some quality attributes of the software is by collecting quality-related dynamic metrics. Collecting dynamic metrics is considered to be very challenging since it usually requires instrumentation of the software. An approach that considerably facilitates the transparent collection of dynamic quality metrics may provide a means to expose some dynamic quality attributes of OSS. In this paper, we discuss how AOP technology can support the collection of dynamic software metrics. The proposed approach is demonstrated by a showcase. The paper is structured as follows: Section 2 summarizes the main results of our survey. Section 3 discusses the introduction of testing frameworks into the testing process of OSS, and the use of AOP technology to support the collection of dynamic quality metrics. We conclude and we draw our future work in Section 4.

2 ZHAO AND ELBAUM: 10 YEARS LATER

2.1 Survey Goals

We had in mind three final goals when we started this survey. First, we want to identify current best practices in testing OSS products and the limits these practices have. We would like to understand whether OSS developers follow the well-agreed testing practices used in CSS development models, and then detect similarities and divergencies between the two practices. Moreover, we would like to understand if there are different testing practices for different application domains. Second, we want to evaluate both the correlation between the popularity of the OSS product and its testing activities, and also the correlation between the testing activities and the bug rate of the product. Last, we want to suggest a set of testing remarks specific for OSS products. In this paper, we start reporting the data we collected by analyzing the testing activity of 33 OSS projects.

Table 1: OSS project classification.

Application Domains	#of Projects
Business Intelligence	6
Middleware	4
Operating System	4
Testing Tool	2
CMS	3
DBMS	2
Development Tool	4
Framework/Library	3
Other	5
Programming Language	**#of Projects**
Java	14
C/C++	15
PHP	3
Assembler	1

We compared the results of our survey with the outputs provided by L. Zhao and S. Elbaum to understand whether the current large adoption of OSS products in industrial environments is influencing the way developers test their OSS products. Ten years ago, OSS products did not have the enjoying increasing popularity and diffusion that they are experiencing in the last few years.

2.2 Data Collection

The first step of our survey was to identify a representative set of OSS products. We focused on active and evolving OSS projects and we limited our universe to projects that are well known in industry and in research. In line with the sample analyzed by L. Zhao and S. Elbaum, we selected 33 OSS products with in mind heterogeneity of the programming language (Java, C, C++, PHP and Assembler) and heterogeneity of the application domain (business intelligence, middleware, operating system, Content Management System, etc...). The complete list of the selected projects can be found in (Qualipso, 2009). Table 1 shows the distribution of the selected projects against the two attributes programming language and application domain.

The second step of our work was to prepare a checklist that could simplify the analysis of each OSS product. The checklist we obtained is composed of 10 entries: (1) project size, (2) time in market, (3) number of developers, (4) user manual availability, (5) technical documentation availability, (6) test suite availability, (7) testing documentation availability, (8) report of test results availability, (9) test domain, (10) and testing framework usage. These entries have been identified and defined with in mind the three goals previously described.

Contrariwise to the work done by Zhao and Elbaum that derive the quality assurance of open source development model by directly interviewing OSS developers, we manually collected the data by surfing the repositories of the selected projects against each entry of the checklist. To avoid errors and oversights in collecting data, we double checked the obtained results by performing the survey with two different working groups. Each working group was composed of two senior researchers. At the end of the survey, the results obtained from the two working groups have been compared and adjusted.

2.3 Survey Results

The data collection process ended with the availability of 96% of the required information. During our surfing process, we were not able to detect (or derive) the size of 8 (out of 33) projects, and the number of developers for 6 (out of 33) projects. Hereafter, we will report these unavailable data as ”`unknown`” data points.

2.3.1 General Descriptive Findings

Each project has been profiled with its project’s size, the number of developers that contribute to its development, and its time in market. We identified the following ranges for these three general descriptive findings:

Project’s Size. Tiny (less than 1000 lines of code), small (1000 - 10000 lines of code), medium (10000 - 100000 lines of code), large (100000 - 1000000 lines of code), very large (more than 1000000 lines of code).

Number of Developers. Small group (less than 10 developers); medium group (11 - 50 developers); large group (more than 50 developers).

Time in Market. Very young (less than 1 year); young (1 - 3 years); mature (3 - 5 years); old (more than 5 years).

In our sampling, the projects are distributed in the size category as follows: 0% are tiny projects, 3% are small, 15.5% are medium, 36% are large, 21.5% are very large, and for 24% of the projects we are not able to detect (or derive) the size of the projects.

12% of the projects are developed by a small group of contributors, 45.5% by a medium group, 24.5% by a large group, and for 18% of the projects we are not able to identify (or derive) the number of the developers.

The vast majority of projects are old or mature projects. Specifically, 3% are young, 30.5% are mature, and 66.5% are old projects.

Most of the projects are large and very large projects with a high maturity and a medium or large community of developers and contributors. As expected, most of the projects started as small projects and tend to evolve during time, and the size of the project depends on the number of the developers. However, in our sampling, we also have two exceptions to these tendencies, where an old project is still small, and a very large project is characterized by a small community of developers. Figure 1 shows the relationship between the project maturity and its size, while Figure 2 highlights the relationship between the number of developers and the size of the project.

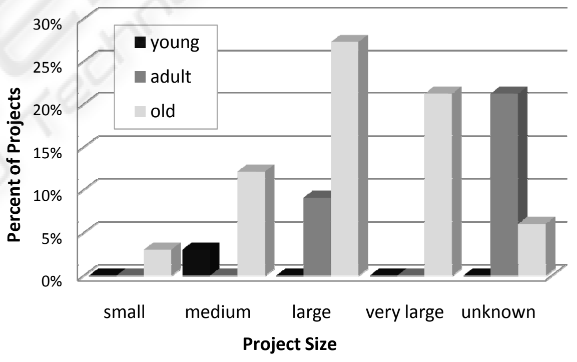

Figure 1: Time in market by project size.

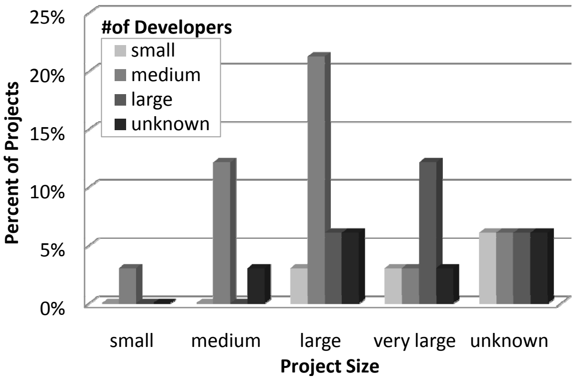

Figure 2: Number of developers by project size.

2.3.2 Evaluated Testing Aspects

Testing Activity. In our survey, we were interested in understanding whether testing is a consolidated activity, or not, during the development of an OSS product. Developers, customers and end users have the perception that testing activities receive less importance in OSS than in CSS. In their survey, Zhao and Elbaum found that 58% of the analyzed projects spent (Zhao and Elbaum, 2003) 20% of the time on testing, while more than 15% of the projects spent more than 40% of their time in testing. In our sampling, we found that 58% of the projects have a test suite or a testing activity, while the remaining 42% does not publish the source code of their testing activities. Specifically, 13 (out of 22) old projects have an updated test suite, while 9 of them do not have a testing activity; 5 (out of 10) adult projects have an updated test suite, 1 project a very preliminary test suite, and 4 projects do not have a testing activity; the young project is released with its test suite. Referring to the project size: the small project has testing activities; all of the 5 medium projects have testing activities; only 6 (out of 12) large projects have a testing activity; and only 4 (out of 7) very large projects have a testing activity. These data confirm the Zhao and Elbaum statement: ”it seems that larger projects tend to spend less time in their testing phase compared with smaller projects.”

Test Planning. The IEEE Standard for Software Test Documentation (IEEE, 1998) defines a test plan as: ”A document describing the scope, approach, resources, and schedule of intended testing activities”. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning. Our investigation has shown that only 15% of the analyzed projects plan the testing activities separately, while 6 projects use the development plan for documenting the testing activities. This can be explained as for these projects the focus is on unit testing which is usually performed by the developers themselves. For the remaining 67% of the 33 investigated projects, we were not able to identify any test planning document. This strongly supports the findings of Zhao and Elbaum (Zhao and Elbaum, 2000) where more than 80% of the OSS developer responded that their products do not have testing plans.

Testing Strategies and Approaches. The Test Strategy defines the strategic plan for how the test effort will be conducted against one or more aspects of the target system. Test Approach is the implementation of the test strategy for a specific project. It typically includes (a) the decisions made that follow based on the (test) project's goal and the risk assessment carried out, (b) starting points regarding the test process, (c) the test design techniques to be applied, (d) exit criteria and (e) test types to be performed. 21 of the investigated projects provide no information about the test strategy and the approach followed to implement it. The other 12 projects provide some information on the test strategy and the test approach. For example, the project ServiceMix (http://servicemix.apache.org) provides recommendations like ”every major code change must have a unit test case”, ”the test must provide as much coverage as possible” or ”every bug filled should have a unit test case”. Though such recommendations provide some information on the test strategy, it does not provide a clear and comprehensive test strategy.

As for this testing aspect, we are not able to provide a comparison with Zhao and Elbaum work, because of the lack of this aspect in their survey.

Testing Levels. Testing can be done at different levels; ranging from testing the individual smallest units of code to testing the completely integrated system. The analysis has shown that only for 1 project three levels of testing are defined, namely: unit, integration and system testing. Unit testing is the preferred activity (16 projects perform unit tests), while acceptance testing is the omitted activity (0 projects have a serious campaign of acceptance tests). This can be explained by the interpretation a lot of developers have regarding unit testing: often, an entire subsystem is wrongly regarded as a unit, or acceptance tests are wrongly mixed with unit tests. Just 1 project has a comprehensive test suite that covers the following testing levels: unit; integration; system testing; and other non-functional tests. In 6 projects, non-functional tests are also considered besides functional tests. In Figure 3, we summarize the distribution of testing levels covered by the investigated projects.

Testing at different levels reduces the costs of repairs since most problems will be exposed early by the lower testing levels (mostly by the developers themselves). Having multiple testing levels increases the chance of uncovering software errors.

Contrariwise to the results of Zhao and Elbaum, where 68% of the respondents ”provide inputs trying to imitate user behavior” and only 25% of them ”use assertions”, our results suggest that the preferred activity is unit testing. This can be explained by the large popularity reached by xUnit frameworks in the last few years.

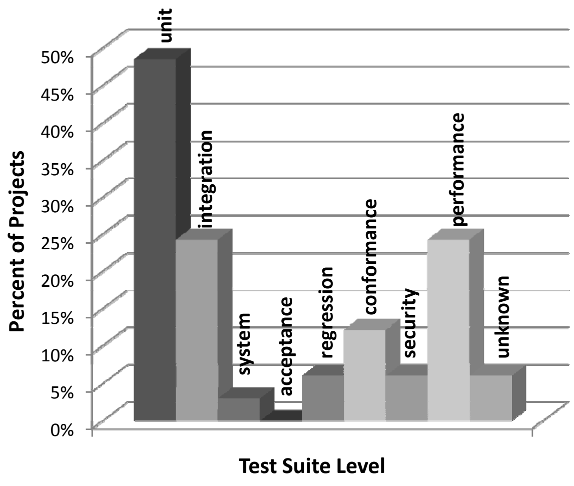

Figure 3: Test level.

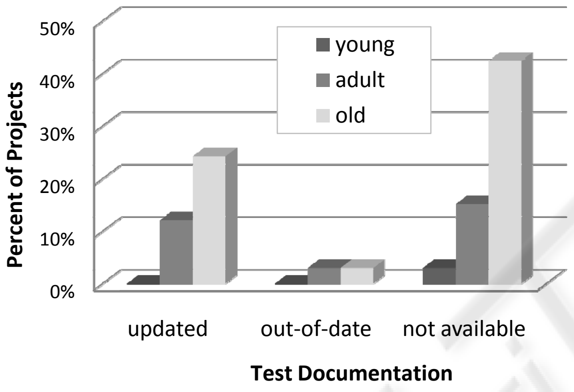

Figure 4: Documentation availability by time in market.

Testing Documentation. Any project document that aims to provide testing related information is considered to be part of the test documentation. This includes test specifications, test design, test procedures, test plans, and test results reports. 8 projects (24%) deliver a report that describes the test results: 7 of them are updated, while one report is out of date. Only 5 projects provide test plans. In general, we discovered 14 testing documents, 12 of which are updated and 2 are out of date. In Figures 4 and 5, we show the relationship between the availability of testing documents and the project maturity, and the availability of testing reports and the project maturity, respectively.

As for this testing aspect, we are not able to provide a comparison with Zhao and Elbaum work, because of the lack of this aspect in their survey.

Testing Tools Support. It is important for a project to control and manage its testing process. A managed testing process increases the overall efficiency of the testing activities. Currently, a lot of tools and

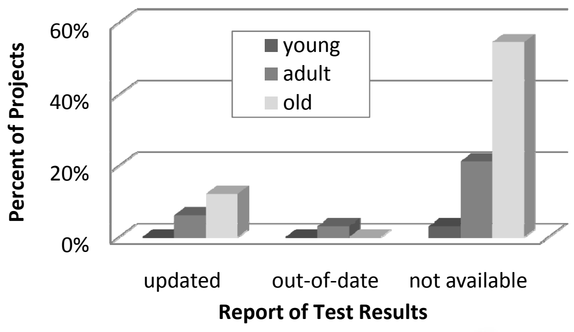

Figure 5: Reports availability by time in market.

plugins are available to support the whole testing process. For example, HP Quality Center [www.hp.com] is a test management tool that covers many aspects of the testing process including: requirements traceability, test planning, test execution, defect management and test reporting. Additionally, the open source Test and Performance Test Tools Platform (TPTP) (www.eclipse.org/tptp) provides partial support for the testing process including test execution, test reporting and test monitoring (in Section 3 we will present how to use TPTP in the context of OSS projects). According to our analysis, we could not identify any project that utilizes a tool supported approach for managing the whole testing process and only 6 projects explicitly use testing frameworks and tools (such as JUnit (www.junit.org) or Apache Test Framework (http://httpd.apache.org/test/)) for their testing activities.

As for this testing aspect, we are not able to provide a comparison with Zhao and Elbaum work, because of the lack of this aspect in their survey.

2.4 Final Remarks

The results we obtained in our survey are in line with the outputs provided by Zhao and Elbaum ten years ago. It seems that larger projects tend to spend less time in their testing phase compared with smaller projects; more than 40% of OSS products do not have a testing activity in their development process; 67% of the 33 investigated projects does not have any test planning document; 36% provides some (often preliminary) information on the test strategy and the test approach; the preferred activity is unit testing; 42% has testing documentation (often incomplete and out-of-date); 18% exploits available testing tools but none of the projects uses a testing framework to support the whole testing process.

The evolution of testing tools, the availability of new testing methods and the increasing necessity of software systems with stringent quality requirements did not change the way OSS developers test their products, in ten years. This is probably due to at least three mutually related reasons: (1) most of the available testing techniques have been defined with closed-source software characteristics in mind, thus they are not directly applicable to OSS systems, so a good deal of effort and cost is required for designing new testing solutions that are created ad-hoc for OSS systems; (2) the planning and the monitoring of the testing process of an OSS system hardly ever follow the guidelines used for CSS systems, so it is necessary to redefine some methods that are at the basis of the testing process; (3) OSS system development hardly ever follows the classic software engineering paradigms found in textbooks, so testing activities are less structured than for CSS.

3 HOW TO REVERSE THIS NEGATIVE TREND?

One of the main challenges with OSS is the absence of comprehensive testing concepts. Most of the OSS is developed with little attention paid to testing and therefore allowing only limited potential for verification. The results presented in this survey showed that for 21 (out of 33) analyzed OSS projects there is no clearly defined testing strategies and approaches. Common testing approaches that are applicable to a wide range of OSS can considerably facilitate OSS testing and verification.

3.1 Framework Support: TPTP

Nowadays, the wide diffusion of testing tools and frameworks that support different testing techniques is evident. A quick look to the portal [www.opensourcetesting.org] can confirm this thesis. The open source community contributed to this by providing plenty of testing tools that are widely accepted within the software testing community. Most testing tools (Open source as well as Closed source) are built to support proprietary data models without giving any consideration for tool integration. Furthermore, most testing tools provide a Graphical User Interface (GUI) that is specifically designed for that tool. This may confuse users dealing with different tools. For the above-mentioned reasons (among many others), the Eclipse foundation started the OSS project TPTP (Test and Performance Testing Tools, previously named Hyades). The main objective of the TPTP project is to build a generic, extensible, standards-based platform for test and performance tracing tools. In this paper, we investigate version 4.4.1 of the TPTP platform and we describe how to apply the framework for testing OSS products. The TPTP is basically a plug-in collection for Eclipse that provides a platform for Quality Assurance (QA) tools. This platform consists of a set of tools and data models for testing purposes, profiling/tracing, logging and monitoring. It also includes an infrastructure for deployment and execution of tests, agents or the like on remote machines and for the collection of information gathered by agents on those machines, like test results, traces, etc... In their initial proposal the TPTP group already outlined a comprehensive concept for widely automated execution of tasks in the quality assurance process, which subsumed the well-known methods of automated test development and execution, under the term Automated Software Quality (ASQ).

The procedure starts by traces gathered for analysis from an application during runtime. Collected traces serve as templates for the development of test cases and scenarios in test models. The effort of this task can be reduced by automated transformation of traces into test descriptions. According to the principle of Model Driven Software Development, the transformation of a test model into executable code can be automated too. The next activity is the deployment of tests on remote machines. Finally, the test is executed where new information on runtime behaviour of the system under test is monitored ("trace"), which might be used for comparison with the original trace achieved in the first activity.

The TPTP project is comprised of four subprojects: (1) the TPTP Platform provides the common infrastructure for the other subprojects; (2) the Testing Tools framework extends the platform with testing capabilities. The subproject provides reference implementations for: JUnit, manual, URL, and automated GUI testing; (3) Tracing and Profiling Tools framework extends the platform with tracing data collection capabilities; (4) Monitoring Tools are targeting the collection, analysis, aggregation, and visualization of the captured data.

Figure 6 presents the overall architecture of the TPTP. The TPTP simply extends the well-known Eclipse IDE with testing capabilities. Therefore, TPTP makes use of Eclipse resources and features. Here, we mainly focus on the Testing Tools subproject, as the other subprojects (tracing, profiling and monitoring tools) are not directly related to testing. The Testing tools subproject is an extension of the TPTP framework that: (1) Provides a common framework for testing tools by extending the TPTP platform; (2) Facilitates the integration of different types of tests by providing common nomenclature and metaphors; (3) Allows the deployment and the execu-

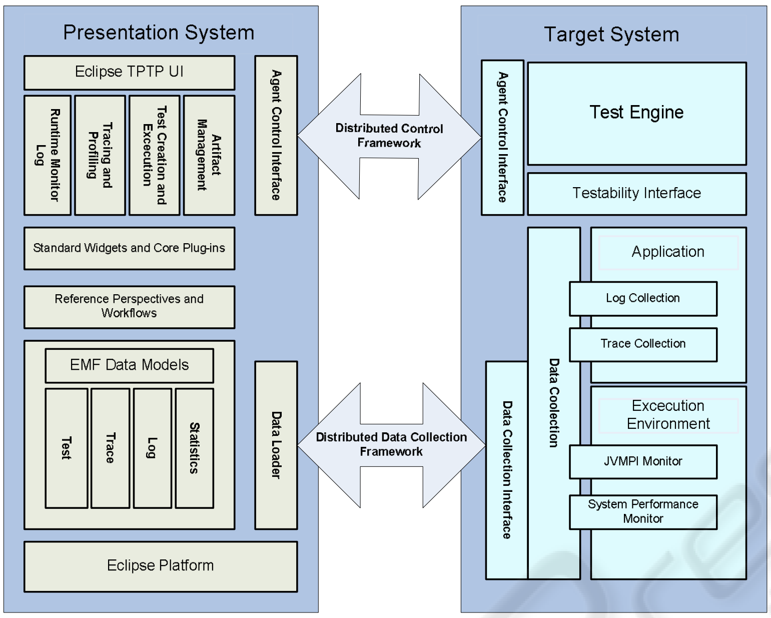

Figure 6: Architecture of the TPTP platform.

tion of tests on distributed systems; (4) Provides common perspective and views.

To demonstrate the potential of the TPTP testing tools framework, TPTP provides exemplary integration of some important testing environments such as: (1) Assisted Manual Testing; (2) JUnit testing framework; (3) Web (URL) Testing; (4) Automated GUI Testing (Recording/Playback). One of the strengths of the TPTP Testing tools framework is the extensibility of the framework to support any third-party testing tool (including commercial testing tools). Third-party tools can be integrated into the platform at different levels: (1) Communications and data collection: The framework provides an agent based framework that supports distributed system communications and communicates the data collected to the data model; (2) Data model: The framework data models are populated by the data collected through the agents. A common data model facilitates the integration of the different tools at data level; (3) Data analysis: The data fed into the data model are analyzed by set of exemplary analyzers. Additional third-party analyzers can be plugged-in into the framework; (4) User interface: Includes test editors, viewers, navigators and wizards. This stack structure tolerates the hook up of any third-party tool at any level. At the same time, the tool plugged-in can still make use of the capabilities of the other levels in the stack.

3.1.1 Using Eclipse TPTP for Testing OSS

Although the eclipse IDE and the TPTP are basically targeting Java applications, the eclipse plug-in CDT (Eclipse C/C++ Development Tooling) provide native support for C/C++ application. Java and C/C++ based OSS projects form around 88% of the surveyed projects in this paper. The eclipse-based TPTP platform can assist establishing testing activities for OSS and tackling some issues raised by the survey like Testing Levels, Testing Documentation and Tools Support. The TPTP platform covers almost all testing levels including unit, integration and system test. The TPTP JUnit Testing Framework primarily targets unit testing level. It can however cover other levels like integration and system testing. Integration testing is usually focusing on ensuring that distinct and individually developed subsystems interface and interact together in a correct and coherent manner. Testing individual subsystems at their interfaces can be done using the API testing technique where the subsystems are tested directly through their APIs and not through their GUIs (if available). The TPTP JUnit Testing Framework can be used to test the subsystems through their APIs. The only difference to the traditional unit testing is that the artefact under test is not an atomic unit of code (e.g., a class), but an entire subsystem. The same technique mentioned above for integration testing can also be applied at system testing level. The artefact under test is here the entire system. For C/C++ applications, the eclipse free and open source plug-ins CUTE and ECUT provide similar functionality to that provided by JUnit for Java applications. Following the argumentation used above for JUnit, CUTE and ECUT can be used not only for unit testing C/C++ applications, but also for integration and system testing. The three unit testing tools JUnit, CUTE and ECUT provide basic support for test case documentation test execution and test result reporting. The TPTP Assisted Manual Testing can be used for system testing where no test automation is possible. As the name implies, it provides some assistance for creating, editing and running manual tests and for the analysis of test results.

3.2 AOP-based Approach for collecting Dynamic Quality Metrics

Software metrics provide methods of determining software quality through quantifying different aspects of software quality. Metrics mostly are defined and collected based on static code analysis. However, the complex dynamic behavior of many applications (e.g. through the usage of OO paradigms like inheritance and polymorphism) only allows a poor predictive accuracy of quality models when using static measurement. Therefore, in addition to static metrics, dynamic ones need to be investigated to evaluate programs according to their dynamic behaviour. Some approaches only consider requirements, like Yacoub et al. (Yacoub et al., 1998) who describe a suite that consists of metrics for dynamic complexity and object coupling based on execution scenarios. The proposed measures are obtained from executable design models.

Other approaches focus on system performance, like Cavarero (Cavarero and Cuaresma, 2004), Wiese's FastAOP (Wiese and Meunier, 2008), Box's Glassbox Inspector (https://glassbox-inspector.dev.java.net/). A set of architecture-independent dynamic metrics can be defined and used to categorize programs according to their dynamic behaviour in five areas: size, data structures, memory use, polymorphism and concurrency. Mitchel and Power (Mitchell and Power, 2005) define a number of Java based object-oriented metrics for dynamically measuring coupling and cohesion at class level. The defined metrics parallel the suite of static object-oriented metrics proposed by Chidamber and Kemerer (Chidamber and Kemerer, 2003) from where two of six metrics, namely coupling and cohesion, were picked because they were thought to be the most useful in providing information about the external quality attributes of a design.

Measuring dynamic metrics is in general more expensive and can be considered more complex than collecting static ones. Many of the approaches apply instrumentation techniques or use debug or profiling interfaces to intercept and gather information (Mitchell and Power, 2005), (Arisholm et al., 2004). The drawback of these approaches is that they tend to be tedious and error prone. As measuring dynamic metrics of an application at runtime is a system-wide crosscutting concern, it seems to be an ideal area of application for aspect oriented programming. In this area, some approaches have already been implemented for Java related applications: FastAOP (http://sourceforge.net/projects/fastaop/) and Glassbox Inspector to measure performance and AOP HiddenMetrics (Cazzola and Marchetto, 2008) to dynamically collect OO metrics.

We developed our own approach for collecting dynamic metrics using AOP and we realized a simple show case. The approach can be summarized in the two following points:

1. AOP Pointcuts are used to identify locations in the program flow where metrics data can be collected;
2. AOP Advices are used to analyze the data collected and provide the metric information.

To test the suitability of the AOP approach, we compiled a show case where we picked an OO metric and collected it dynamically for one of the Java OSS projects considered for the analysis.

3.2.1 Selected Metric

Measuring metrics dynamically is necessary to evaluate programs according to their dynamic behaviour. Coupling and cohesion metrics can serve as key indicators for evaluating the complexity of an application and thus can be used to gain insights when assessing quality factors. For our show case, we picked the Coupling on Method Call (CMC) metric (Cazzola and Marchetto, 2008) to measure coupling on method calls. As we want to perform measurements dynamically, we would like to find out at runtime how many classes a class A is coupled to.

3.2.2 Project Identification

We applied this metric to Apache JMeter (http://jakarta.apache.org/jmeter/), a desktop application designed to load test functional behavior and measure performance. It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types. We used the JMeter version 2.3.2, the one analyzed in our survey of Section 2.

3.2.3 CMC Aspect Definition

The metric measurement has been implemented using AspectJ [www.eclipse.org/aspectj/] by defining join points on method calls and executions (to catch the inheritance hierarchy, too) and defining join points for excluded packages like java.* or org.aspectj.* (see Listing 1).

Advices are then used to analyze the information at the join points. By means of reflection (`thisJoinPoint`, `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart`) the necessary data (caller, callee) are extracted and stored if calls from one class to another are concerned. The gained information is held in a hashtable and at the end of the program execution sorted for final output (see Listing 2).

3.2.4 Experimental Setup

One practical drawback in using dynamic analysis is that one has to ensure that the code is sufficiently exercised to reflect program execution in a complete or at least a typical manner. For our show case, we set up a JMeter test plan for a typical stress testing scenario according to the step-by-step user manual of JMeter, where a stress test can be generated out of the recording of visited websites. After that the access to the target websites is recorded by JMeter, the user has to configure the test parameters like number of threads and loop count to complete the test plan. For our show case, we start the JMeter test from a Java program using `org.apache.jmeter.NewDriver.main(...)` and the created test plan as parameter.

For the development of the show case, Eclipse version 3.3.2 [http://www.eclipse.org/eclipse] was used in combination with AspectJ Development Tools (AJDT) version 1.5.1 [http://www.eclipse.org/ajdt]. The AJDT project provides Eclipse platform based tool support for aspect oriented software development with AspectJ and is installed as an Eclipse plug-in. This environment was also used as runtime environment for the show case.

The CMC aspect has been implemented using AspectJ. We applied it to the JMeter test execution scenario through load time weaving. This can be achieved within Eclipse through the run dialog where the "AspectJ Load-time Weaving" configuration has to be chosen. In the LTW Aspectpath tab the project containing the aspect has to be added. Additionally, it is required to set the working directory (arguments tab) to /bin of the JMeter installation folder as JMeter is hard coded to look up jmeter.properties in this folder. After the setup is completed, the test can be started and the CMC aspect will be woven into the JMeter libraries at load time. This enables us to collect the metric during runtime.

```
/**
 * Join points on excluded methods.
 */
pointcut excluded():
  cflow(call(* java..*(..))) ||
  cflow(call(java..*.new(..))) ||
  cflow(call(* org.aspectj..*(..))) ||
  within(CMC);
/**
 * Join points on method calls.
 */
pointcut method_calls():
   (call(* *..*(..)) ||
    call (*..*.new(..))) &&
    within(org.apache.jmeter..*);
```

Listing 1: Definition of join points for CMC.

```
/**
* Advice to analyze method calls.
*/
before(): method_calls() && !excluded() {
 String caller =
 thisEnclosingJoinPointStaticPart.getSignature().
      getDeclaringTypeName();
 String callee =
 thisJoinPointStaticPart.getSignature().
      getDeclaringType().getName();

//update hashtable if caller and callee refer to
     different classes
  if (!caller.equals(callee)) {
      updateCouples(caller, callee);
  }
}
```

Listing 2: Advice to analyze method calls.

3.2.5 Results

The output from our show case simply is a list of all the classes that are coupled to other ones and the number of classes they are coupled to (see Listing 3).

The experimental setup as described above has shown that AOP is a feasible and elegant way to support dynamic measurements in order to collect metrics at runtime. The AOP code for this show case was tailored to fit exactly the JMeter test scenario. To make use of this approach in a more general way, abstract pointcuts can be introduced, for example, that can then be extended to fit a particular application that should be measured. Moreover, the measuring support can easily be extended for other metrics by adding appropriate aspects respectively.

```
org.apache.jmeter.JMeter$ListenToTest          3
org.apache.jmeter.NewDriver                    2
org.apache.jmeter.config.Arguments             4
org.apache.jmeter.config.ConfigTestElement     1
org.apache.jmeter.control.GenericController    3
org.apache.jmeter.control.LoopController       3
org.apache.jmeter.engine.PreCompiler           6
org.apache.jmeter.engine.StdJMeterEngine      16
org.apache.jmeter.engine.
               StdJMeterEngine$StopTest        1
...
```

Listing 3: The CMC metric result for JMeter.

4 CONCLUSIONS

Our survey has shown that more than 40% of OSS products do not have a testing activity in their development process. The availability of test suites, or more in general the presence of testing activities, is independent from the maturity of the OSS product. This is probably due to the recent explosion of tools that support testing activities. Young OSS products can exploit these tools in the same way mature products can do. In particular, the survey points out that systematic acceptance, system, and regression testing activities are marginally exploited in OSS. This is probably due to the wrong practice of mixing acceptance and system tests in unit test cases. Regression testing is probably complicated by unstructured teams that contribute to the project with small and disaggregated pieces of code.

All these considerations suggest the adoption of tools that can support the whole testing process, starting from the plan of the testing activities to the report of test results. In this paper, we have deeply described the TPTP framework and how to exploit its potentialities in the context of OSS projects.

Testing is mainly concerned about validating the dynamic behaviour of the software. In this paper, we have also introduced the idea of exploiting the potentiality of AOP technology to support testing. AOP can actually help exposing the dynamic behaviour of the software in terms of dynamic metrics.

ACKNOWLEDGEMENTS

The research presented in this paper has been partially funded by the IST project (http://www.qualipso.eu/), sponsored by the EU in the 6th FP (IST-034763) and the FIRB project ARTDECO, sponsored by the Italian Ministry of Education and University.

REFERENCES

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

Cavarero, J. and Cuaresma, M. (2004). Metrics for dynamics: How to improve the behaviour of an object information system. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*, pages 344–349.

Cazzola, W. and Marchetto, A. (2008). Aop hiddenmetrics: Separation, extensibility and adaptability in sw measurement. *Journal of Object Technology*, 7(2):53–68.

Chidamber, S. and Kemerer, C. (2003). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493.

IEEE (1998). IEEE standard for software test documentation: IEEE std. 829-1998.

Lewis, W. (2004). *Software Testing and Continuous Quality Improvement*. Auerbach, second edition.

Mitchell, A. and Power, J. (2005). Using object-level runtime metrics to study coupling between objects. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1456–1462.

Qualipso (2009). Web published: www.qualipso.eu/node/129. Accessed: September 2009.

Wiese, D. and Meunier, R. (2008). Large scale application for aop in the healthcare domain: A case study. In *Proceedings of the Aspect Oriented Software Conference (AOSD)*.

Yacoub, S., Ammar, H., and Robinson, T. (1998). Dynamic metrics for object oriented designs. In *Proceedings of the IEEE International Symposium on Software Metrics*, pages 50–61.

Zhao, L. and Elbaum, S. (2000). A Survey on quality related activities in OS. *ACM Software Engineering Notes*, 25(2):54–57.

Zhao, L. and Elbaum, S. (2003). Quality assurance under the open source development model. *International Journal of Systems and Software*, 66(1):65–75.