QualiPSo

Quality Platform for Open Source Software

IST- FP6-IP-034763

Working Document 5.4.2

Test suites and benchmarks for the chosen set of Open Source projects and artifacts. Methodology for creating test suites and benchmarks for arbitrary systems.

Auri Marcelo Rizzo Vincenzi (USP)
Davide Taibi (INS)
Davide Tosi (INS)
Giovanni Denaro (INS)
Leonardo Mariani (INS)
Marcio Eduardo Delamaro (USP)
Marcos Lordello Chaim (USP)
Sandro Morasca (INS)

Due date of deliverable: 31/07/2010

Actual submission date: 31/07/2010

This work is partially funded by EU under the grant of IST-FP6-034763.

Change History

Version	Date	Status	Author (Partner)	Description
2	July, 27 2009	Draft	Marcos L. Chaim	First version of the draft
2	August, 3 2009	Draft	Davide Tosi and Marcos L. Chaim	**T-DOC** included in the first version of the draft
2	October, 27 2009	Final	Davide Tosi and Marcos L. Chaim	Comments of the reviewer were addressed by the authors.
3	May, 31 2010	Draft	Davide Tosi and Marcos L. Chaim	Added example of T-DOC with Macxim
3	May, 31 2010	Draft	Davide Tosi and Marcos L. Chaim	Comments from the revision addressed
3	July, 30 2010	Draft	Davide Tosi and Marcos L. Chaim	RealEstate size and coupling metrics added
3	July, 30 2010	Draft	Auri Vicenzi and Marcos L. Chaim	Additional OSS project testing coverage add
3	July, 31 2010	Final	Davide Tosi and Marcos L. Chaim	Minor corrections

Executive Summary

We present in this document three different assessments of the testing activity performed in well-regarded OSS projects: 1) coverage evaluation; 2) maturity level of testing process; 3) availability of testing documentation.

The first one consisted of evaluating the coverage with respect to structural testing criteria provided by the test suites developed in OSS projects. Control- and data-flow based criteria were utilized. Eight OSS projects were analyzed, namely, HSQLDB, HTTPUnit, JasperReports, JMeter, JUnit, Log4J, PMD and Velocity. Data shows a modest coverage for HSQLDB, JasperReports, JMeter, Log4J, and PMD while JUnit and Velocity coverage data is around 50%. The notable exception is HTTPUnit whose coverage level is above 70%.

We also collected similar control-flow coverage metrics and dynamic coupling metrics for a particular example – the OSS RealEstate Java [application NCSU, 2009]. To collect the control-flow coverage a different tool was utilized and to collect the dynamic coupling metrics Aspect-oriented Programming (AOP) techniques were used. The results of the control-flow metrics show that less than 50% of the Source Code Lines of Code (SLOC) and of the methods were exercised by the available test suite. This data is in line with the data collected for the former OSS projects. We could observe that the dynamic coupling metrics were easily collected with the AOP resources.

Hence, for the majority of the analyzed projects, the test suites need to improve. One possible hypothesis for this behaviour is that many tests are *not* added in the test suites. A developer may fix or add a new feature, create a test to verify it, but not add it to the test suite. Furthermore, much of the testing being carried out in the OSS context is expected to be performed by final users. They might explore the code of the OSS product but their tests are not registered in test suites.

Higher statement (nodes) and decision (edges) coverage has been advocated for Closed-Source Software (CSS). [Beizer 1990] states that node and edge coverage criteria are the weakest structural testing criteria; though, he complements that “testing less than this for new software is unconscionable and should be criminalized.” Formal standards like DO-178B [RTCA 1992] and ANSI/IEEE 1008-1997 [IEEE 1987] demand 100% statement and branch coverage for safety critical systems. And [Cornett 2009] discussing about the minimum acceptable code coverage argues that a 70-80% coverage level is a reasonable goal for system testing of the majority of software products.

With the exception of a single project, the analyzed OSS projects test suites need to be improved to reach the 70%-80% level, suggesting that the process to develop them differ from CSS products process and requires different ways of evaluating and improving its trustworthiness.

The second assessment consisted of evaluating the level of maturity of the OSS testing process. To do so, a novel maturity model targeted to OSS testing is presented. The first task of the OSS testing maturity model (OSS-TMM) is to identify the “Best Testing Process” (BTP) for a particular OSS project. The BTP

is identified by inspecting the project according to a checklist developed to guide the elicitation of tasks needed to achieve the OSS trustworthiness. Then the “Actual Testing Process” (ATP) of the OSS project is obtained by checking the testing tasks actually executed by the community supporting the project. Depending on the level of compliance of the ATP with respect to the BTP, a level of maturity is assigned to the OSS project.

To demonstrate its applicability, OSS-TMM was utilized to analyze in detail two real-life projects, BusyBox and Apache HTTP. In addition, four more representative OSS projects were assessed with OSS-TMM in order to correlate their maturity levels with their bug rates to comprehend whether a higher maturity of the testing process directly means a higher product quality. Our observations are that OSS-TMM can be easily applied either to small or to large OSS projects, but the correlation between the level of maturity and bug rates was verified only partly. OSS-TMM has been also applied internally to Siemens AG to evaluate the testing maturity level of TPTP (The Eclipse Test and Performance Tools Platform). The evaluation has been carried out both collecting information stored into the TPTP repository and by interviewing TPTP project leaders and developers.

The third assessment carried out was the analysis of 32 OSS projects about the availability of testing documentation. We have observed a remarkable deficiency with regards to testing documentation. In our analysis, only one product provides a complete documentation about internal testing activities. The difficulty of providing an up-to-date and a reasonable documentation of OSS products relates to two main reasons: first, documenting development activities and technological issues is a tedious and unrewarding task; second, data and information about the OSS project (such as source code, project plans, testing requirements, etc.) are disaggregated and shared via unstructured channels such as unofficial forums and mailing lists.

In this document, a framework (called T-DOC) that supports the automatic generation of test cases documentation, the generation of reports about test case results, and the archiving of testing documents in central repositories is proposed. The automatic generation of documentation is facilitated by the adoption of built-in testing methodologies that simplify the aggregation of testing data. By introducing T-DOC, we aim at addressing the lack of documentation in OSS projects due to the use of external testing methodologies that drastically augments the fragmentation of data. We applied the framework to the OSS RealEstate Java application to show the applicability and the real benefits of our solution.

Thus, this document presents both experimental results on the assessment of the source code provided by OSS projects test suites as well as techniques to assess and improve the testing process in the context of OSS development.

These assessments and techniques have implications to different stakeholders.

- From a software company's point of view:
    - The assessment of the OSS test suite coverage shows how thorough the testing of the OSS functional requirements was;

  - The application of OSS-TMM simplifies the internal process of testing OSS products by suggesting a rapid way for identifying a testing plan that best fits the properties and characteristics of the OSS product; it simplifies the assessment or certification process of their OSS products by comparing their available testing;
  - The use of T-DOC favours the creation of testing documentation that contributes to assess the trustworthiness of an OSS.

- From the developer's point of view:
  - The assessment of the OSS test suite coverage shows how much effort is needed to achieve an established level of coverage;
  - The application of OSS-TMM simplifies and speeds up testing activities by guiding developers in selecting testing strategies and methodologies depending on the properties and characteristics of the OSS product; it increases the quality and the trustworthiness perception of the OSS by improving the testing activity.
  - T-DOC facilitates the production of a comprehensive testing documentation that originally comes from distributed developers.
- From the end-user's point of view:
  - The coverage information is one element which allows the end-user to make an informed decision on using or not using the OSS;
  - The application of OSS-TMM simplifies and speeds up the selection of an OSS product by evaluating the maturity of the testing process as an evaluation element about the whole quality and trustworthiness of the product;
  - The T-DOC automatic generated documentation allows the end-user to assess the testing activity performed in OSS project.

Document Information

IST Project Number	FP6 – 034763	**Acronym**	QualiPSo
Full title	Quality Platform for Open Source Software		
Project URL	http://www.qualipso.org		
Document URL			
EU Project officer	Michel Lacroix		

Deliverable	**Number**	5.4.2	**Title**	Test suites and benchmarks for the chosen set of Open Source projects and artifacts. Methodology for creating test suites and benchmarks for arbitrary systems.
Work package	**Number**	5.4	**Title**	Definition of standard test approaches, test suites, and benchmarks of Open Source Software
Activity	**Number**	5	**Title**	Trustworthy Results

Date of delivery	**Contractual**		**Actual**	
Status	Version 3.0, dated 31/07/2010		final ☐	
Nature	Report ☑ Demonstrator ☐ Other ☐			
Dissemination Level	Public ☑ Consortium ☐			
Abstract (for dissemination)	We present three different assessments of the testing activity performed in well-regarded OSS projects based on: 1) coverage evaluation; 2) maturity level of testing process; and 3) availability of testing documentation. The first one reveals that, for the majority of evaluated projects, the coverage with respect to structural testing criteria needs improvement, which suggests OSS projects utilize a different testing process requiring different ways of trustworthiness evaluation and improvement. The second assessment utilizes a novel maturity model targeted to OSS testing–the Open-source Software Testing Maturity Model (OSS-TMM). We utilized OSS-TMM in several projects to evaluate their testing maturity, to assess the OSS-TMM ease of applicability, and to correlate testing maturity and OSS trustworthiness. Our final assessment shows that low effort is directed towards developing testing documentation in OSS projects. To tackle this situation, we introduce the TDOC framework which objective is to support a team of OSS developers in creating test documentation that will enhance OSS trustworthiness.			
Keywords				

Authors (Partner)	Auri Marcelo Rizzo Vincenzi (USP), Davide Taibi (INS), Davide Tosi (INS), Giovanni Denaro (INS), Leonardo Mariani (INS), Marcio Eduardo Delamaro (USP), Marcos Lordello Chaim (USP), Sandro Morasca (INS)			
Responsible Author	Auri M R Vincenzi		**Email**	auri@inf.ufg.br
	Partner	USP	**Phone**	+55 (62) 3521-1181

Table of Contents

List of Figures

List of Tables

Differences with respect to the previous version

Version v3 is an improvement on version v2 of the Working Document 5.4.2. It contains additional testing coverage data collected using JaBUTi for four extra OSS projects, namely, HTTPUnit, JasperReports, Log4J, and Velocity. The results obtained from these extra OSS projects corroborate the observation contained in v2 in which the test suites provided by OSS project do not thoroughly assess the structural coverage of the code, indicating that different ways to assess the testing activity in the realm of Open-source development is in need. However, there was a notable exception. HTTPUnit test suite obtained coverage expected for CSS projects.

Additionally, we collected similar control-flow coverage metrics and dynamic coupling metrics from the OSS RealEstate Java application [NCSU, 2009]. To collect the control-flow coverage a different tool was utilized and to collect the dynamic coupling metrics Aspect-oriented Programming (AOP) techniques were used. The results of the control-flow metrics show that less than 50% of the Source Code Lines of Code (SLOC) and of the methods were exercised by the available test suite. This data is in line with the data collected for the previous OSS projects.

Finally, we present in this new version a case study of the application of T-DOC in the development of MACXIM—a tool developed in the context of Working Package A5 for metrics collection from Java programs. Data regarding the improvements in the testing process by using T-DOC as well the automatically documentation generated by it are presented.

INTRODUCTION

1.1. Background

Closed source software (CSS) testing is a well established activity. Unit, integration, and acceptance testing are expected to be performed to assess its functional requirements [Pfleeger 2009]. White- and black-box [Beizer 1990] [Myers 2004] and fault-based techniques [Budd, et al. 1980] have been devised to support the development and assessment of test suites. Formal standards for using white-box techniques have been established depending on the CSS area of application.

The DO-178B standard for certifying safety critical software [RTCA 1992] and ANSI/IEEE 1008-1997 [IEEE 1987] demand 100% statement and branch coverage. A less formal recommendation [Cornett 2009] suggests that the minimum acceptable code coverage level should vary from 70-80% for the majority of software products. Moreover, Cornett also argues that unit, integration and system testing levels demand a decreasing coverage level since, in general, it is easier to achieve a higher coverage of a single unit than of an entire system.

Non-functional requirements should also be assessed in CSS projects. The testing activity is also responsible to verify that a software is adherent to its non-functional requirements. It is equally important to assert that a software system is able to meet its performance requirements, to cope with workloads close to its defined limits, and perform acceptably when its limits are surpassed. Thus CSS is expected to go through stress, load, usability, and configuration testing before being deployed to the user's site or released [Pfleeger 2009].

At first sight, one should not expect much difference of CSS and Open-Source Software (OSS) testing since any software needs to have its functional and non-functional requirements assessed. In the QualiPSo Project Working Package 5.4 – "Definition of standard test approaches, test suites, and benchmarks of Open Source Software", version 1 [Qualipso 2009], testing artefacts developed in well-regarded OSS projects were investigated. Artefacts such as test plans, unit, integration, system and acceptance tests, test scripts, and test reports were searched. Although the majority of OSS projects provide scripts for automating test case execution using tools such as Ant and Maven, artefacts denoting a systematic approach for OSS testing were not identified. According to this investigation, tests in the OSS context seem to be carried out in a rather *ad hoc* fashion [Qualipso 2009].

We believe that this is primarily due to at least three mutually related reasons: (1) some testing techniques that are well agreed on for CSS 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 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 methodologies that are at the basis of the testing process; (3) OSS

system development hardly ever follows the classic software engineering paradigms found in textbooks, but it is closer to Agile and XP development paradigms, so testing activities and testing processes are less structured.

Nevertheless, when OSS use is advocated in place of CSS, a fair question is whether OSS is as trustworthy as CSS. For an end-user it does not matter whether a product is open- or closed-source software because she is interested in dependable products that supply her needs. In this document, we present three different assessments of the testing activity performed in well-regarded OSS projects. The first one consisted of evaluating the coverage with respect to structural testing criteria provided by the test suites developed in OSS projects. Control- and data-flow based criteria [Rapps and Weyuker 1985] were utilized. Eight OSS projects were analyzed, namely, HSQLDB, HTTPUnit, JasperReports, JMeter, JUnit, Log4J and PMD. Data show a modest coverage for HSQLDB, JasperReports, JMeter, Log4J, and PMD while JUnit and Velocity coverage data is around 50%. The notable exception is HTTPUnit whose coverage level is above 70%. Hence, the majority of analyzed OSS projects test suites need to be improved to reach the 70%-80% level, recommended for CSS.

The second assessment consisted of evaluating the level of maturity of the OSS testing process. To do so, a novel maturity model targeted to OSS testing is presented. The first task of the OSS testing maturity model (OSS-TMM) is to identify the “Best Testing Process” (BTP) for a particular OSS project. The *BTP* is identified by inspecting the project according to a checklist developed to guide the elicitation of tasks needed to achieve OSS trustworthiness. Then the “Available Testing Process” (ATP) of the OSS project is obtained by checking the testing tasks actually executed by the community supporting the project. Depending on the level of compliance of the *ATP* with respect to the *BTP*, a level of maturity is assigned to the OSS project.

To demonstrate its applicability, OSS-TMM was utilized to analyze in detail two real-life projects, namely, BusyBox and Apache HTTP. In addition, four more representative OSS projects were assessed with OSS-TMM in order to correlate their maturity levels with their bug rates to comprehend whether a higher maturity of the testing process directly means a higher product quality. Our observations show that OSS-TMM can be easily applied either to small or to large OSS projects, but the correlation between the level of maturity and bug rates was verified in part.

An essential aspect of the testing process maturity is the documentation of the testing activity performed in an OSS project. The third assessment carried out was to analyze 32 OSS projects about the availability of testing documentation. We have observed a remarkable deficiency with regards to testing documentation. In our analysis, only one product provides a complete documentation about internal testing activities. The difficulty of providing an up-to-date and a reasonable documentation of OSS products relates to two main reasons: first, documenting development activities and technological issues is a tedious and unrewarding task; second, data and information about the OSS project (such as source code, project plans, testing requirements, etc.) are disaggregated and shared via unstructured channels such as unofficial forums

and mailing lists.

In this document, we also focus on the problem of documenting testing activities and we propose a framework (called T-DOC) that supports the automatic generation of unit, integration, regression testing documentation, the report of test results, and the aggregation of these data in dedicated central repositories called "testing tracker systems". The automatic generation is simplified by the use of built-in testing methodologies that put together the code of methods and test cases in a single component to avoid the fragmentation of source code and to simplify the aggregation of the testing data [Beydeda 2005]. Moreover, our framework provides a three-layers support, starting from (1) the automatic generation of test cases documentation (in a java-doc like style), (2) the automatic generation of suggestions about integration and regression testing activities that should be performed by each developer and for each component of the project; (3) the automatic generation of reports about the results of the test suite execution. All the documents and testing data are then collected and archived in the testing tracker system of the project to favor data discovery and data sharing.

We apply an initial implementation of the T-DOC framework to the RealEstate Java [NCSU, 2009] application to show the simple applicability, the real benefits and the level of automatization provided by our solution. The documentation generated for this application is presented in Appendix B.

1.2. Objectives

The overall goal of the QualiPSo project is to define and implement technologies, procedures and policies to leverage the OSS development current practices to sound and well recognised and established industrial operations. Activity 5 of the QualiPSo project is mainly concerned with the trustworthiness of the OSS. The main goal of Activity 5 is the identification, quantification, and assessment of the quality factors related to the software products as well as to the artefacts produced during software development that affect trust in OSS products, with emphasis on functional and non-functional factors. This will lead to a quantitative body of knowledge and a set of criteria for establishing trust in OSS.

In this version of this working document, we aim at presenting different ways to assess the quality of pre-existent test suites and also techniques which can be used to evolve test suites and testing processes of OSS projects. These assessments and techniques have implications to different stakeholders.

- From a software company's point of view:
    - the assessment of the OSS test suite coverage shows how thorough the testing of the OSS functional requirements was;
    - the application of OSS-TMM simplifies the internal process of testing OSS products by suggesting a rapid way for identifying a testing plan that best fits the properties and characteristics of the OSS product; it simplifies the assessment or certification process

of their OSS products by comparing their available testing;

- The use of T-DOC favours the creation of testing documentation that allows a company to assess the trustworthiness of an OSS.

- From the developer's point of view:
  - The assessment of the OSS test suite coverage shows how much effort are needed to achieve an established level of coverage;
  - The application of OSS-TMM simplifies and speeds up testing activities by guiding developers in selecting testing strategies and methodologies depending on the properties and characteristics of the OSS product; it increases the quality and the trustworthiness perception of the OSS by improving the testing activity.
  - T-DOC facilitates the production of testing documentation by fragmented developers.
- From the end-user's point of view:
  - The coverage information is one element which allows the end-user to make an informed decision on using or not using the OSS;
  - The application of OSS-TMM simplifies and speeds up the selection of an OSS product by evaluating the maturity of the testing process as an evaluation element about the whole quality and trustworthiness of the product;
  - The T-DOC automatic generated documentation allows the end-user to assess the testing activity performed in OSS projects.

1.3. Structure

The remainder of this document is divided into the following sections. Section 1.3 presents the empirical evaluation of four OSS projects test suites according to four structural testing criteria. Section 2.6 describes details of the OSS testing maturity model (OSS-TMM) and of the experience in applying OMM-TMM on BusyBox and Apache HTTP, and reports on the maturity level of four additional OSS projects. Section 3.7 presents the T-DOC framework. Finally, Section 4.4 concludes this work by summarizing the achieved results.

2. Empirical evaluation of OSS projects' test suites

In this section, we present the results obtained from the application of structural testing to assess a set of OSS projects. This initiative is part of our objectives in the context of the QualiPSo project, in an attempt to identify the current state of practice of the OSS community while developing test sets for OSS. We utilized four structural testing criteria—namely, all-Nodes, all-Edges, all-Uses, and all-Potential-Uses—to assess the thoroughness of functional requirements testing in OSS projects.

To conduct the coverage analysis of the OSS projects we used JaBUTi – Java Byte-code Understanding Tool [Vincenzi, et al. 2005] — a tool that statically analyzes bytecode compiled programs and obtains testing requirements with respect to the aforementioned testing criteria. In addition, JaBUTi instruments the analyzed byte-code so that the testing requirements are tracked at run-time to produce the coverage report with respect to the testing criteria.

We also investigate similar control-flow coverage and dynamic coupling metrics in the OSS RealEstate Java application. The idea is to verify the data obtained for similar control-flow coverage metrics and to have a preliminary assessment of dynamic coupling metrics behavior. The coverage metrics were collected with a different tool [EclEmma, 2010] and the coupling metrics were obtained using AOP resources.

We start off with a brief description of the criteria utilized and of JaBUTi, as well as details of the experiment conducted with this tool. In what follows we discuss the metrics collected in the OSS RealEstate Java application and their results.

2.1. Structural testing criteria

In structural testing techniques, product implementation aspects are crucial for choosing test cases. The term "structural" is related to the knowledge of the internal structure of product implementation. Structural testing criteria are in general classified as follows:

- *Control-flow based testing criteria*: only characteristics of the execution control of a product implementation, such as statements and edges, are used to determine the necessary testing requirements. The most well-known criteria are: *All-Nodes* – requires the execution of all statements of a product implementation at least once; and *All-Edges* – requires a test set that makes each conditional statement assume true and false values at least once.

- *Data-flow based testing criteria:* information about the data flow in the program is used to determine testing requirements. Such criteria explore the interaction between the value assignment of variables and further references to establish the set of testing requirements. The best known data-flow testing criterion – *All-Uses* [Rapps and Weyuker 1985] – requires a test set

T to include tests that exercise paths without redefinitions of a variable *X* from every definition of *X* (a value assignment to *X*) to every subsequent use of *X* (a reference to *X*) (such paths are called def-clear paths with respect to *X*). The *All-Potential-Uses* [Maldonado 1991] criterion is variation of All-Uses in which the test set *T* should include tests that exercise def-clear paths from every definition of *X* to any point of the program reachable by a def-clear path with respect to *X*. The idea is to check potential uses of *X*.

2.2. JaBUTi

To support the application of the structural testing criteria presented, we have been working on the development of an Open Source testing tool called JaBUTi [Vincenzi, et al. 2005].

We have worked on this tool since 2004, improving its functionalities and extending its application to a variety of software products. Currently, besides testing Java programs at unit level, the tool may also be applied for unit testing of Aspect-Oriented programs, Java components, Java micro-edition, and mobile programs, among others. In addition, the tool can be easily employed to work with any language which generates bytecode as a result of the compilation process.

All these variations of JaBUTIi share the core of the tool, which has general functionalities for structural testing. This core is responsible for: static analysis of the system under test (SUT), computation of the required elements, program instrumentation, execution of the instrumented program, and coverage analysis.

A special feature of JaBUTi is to explicitly determine and track at run-time structural testing requirements associated to exception handling structures. JaBUTi divides structural requirements in *exception independent* (ei) and in *exception dependent* (ed) requirements. Thus the control- and data-flow based testing criteria are divided in the following sets: All-Nodes-exception-independent; All-Edges-exception-independent; All-Uses-exception-independent; All-Potential-Uses-exception-independent; All-Nodes-exception-dependent; All-Edges-exception-dependent; All-Uses-exception-dependent; and All-Potential-Uses-exception-dependent.

The *ei* set requires testing the control- and data-flows associated to the main execution path of a program whereas the *ed* set addresses the testing of flows associated to the handling of exceptions. By summing up both the *ei* and the *ed* sets one obtains the total coverage with respect to a particular testing criterion.

2.3. Empirical evaluation

Our first task was to make a static evaluation of some open source projects, namely HSQLDB, JUnit, JMeter, PMD, Weka, ServiceMix, Talend Open Studio, SpagoBI, Cimero, Jboss Application Server, Mondrian, Pentaho, and Spago. We have concluded that all of them have, in general, unit test sets (JUnit like)

associated with them and, as they are integrated with automated tools (Ant or Maven), it can be assumed that they are often run. However, despite this testing culture, the testing techniques and levels applied by the OSS development community could not be identified with accuracy. Considering the current state of testing carried out by OSS communities, it can be observed that:

- In general, the only testing criterion applied is functional; other types of testing like performance and load testing are not available for the majority of the OSS projects. There is no clear evidence of structural (control, data-flow) or fault based testing.
- There is no clear distinction between unit, integration, and system testing. Although there are test suites integrated into the build process (most projects use Ant or Maven to manage software compilation and packaging), there are no clearly defined test plans and strategies after the execution of the test suite–for example, how to proceed when failed test cases are found (e.g. if more than 10% of the test cases failed, the developers must be notified and the software package cannot be released).

A question which regards these test suites is: “Are *ad hoc* test suites sufficient to assign trustworthiness to OSS?” To answer this question we use an approach which comprises structural testing criteria for test set evaluation and further evolution.

Even though Beizer [Beizer 1990] declares that statement and branch coverage criteria are the weakest structural testing criteria, he complements that “testing less than this for new software is unconscionable and should be criminalized”. Regardless of the level of coverage obtained, the importance of coverage testing does not lie on identifying which parts of the product were exercised during test set execution, but on identifying the ones which have *not yet been executed*.

Our intention is to perform the evaluation on all the aforementioned OSS projects. We have current data for eight OSS projects presented in Table 1. The evaluation is performed using the JaBUTi testing tool, and accounts for all its supported criteria. The authors may be contacted to provide full data regarding the experiment.

Table 1 – Description of OSS projects

Name	Version	Description	Project Homepage
HSQLDB	1.9 Alpha 2	Lightweight SQL Database Engine	http://hsqldb.org
HTTPUnit	1.7R1024	Emulates the relevant portions of browser behavior for test automation	http://httpunit.sourceforge.net/
JasperReports	3.5.3R2881	Open source reporting engine	http://jasperforge.org/
JMeter	2.3.2	Load test functional	http://jakarta.apache.org/jmeter/

		behavior and measure performance tool	
JUnit	4.9	Unit testing framework	http://www.junit.org
Log4J	1.2R791506	Logging of application behavior	http://logging.apache.org/log4j/
PMD	5.0	Java source code problem detection	http://pmd.sourceforge.net/
Velocity	1.6.2R791506	Template engine that can be used for many purposes: Web applications, Source code generation, Automatic e-mails and XML transformation	http://velocity.apache.org/

The OSS projects are implemented in Java and correspond to the release mentioned in Table 1. Our first evaluation consisted in identifying the size of the projects and the characteristics in terms of the number of classes and methods (divided in those which do not employ exception handling constructs and those which do).

Table 2 shows, for each project, the size in terms of number of bytecode instructions (Size), lines of source code (LOC, extracted from the bytecode information), number of classes, number of classes with exception handling constructions (and its percentage with respect to the total), number of methods, and number of methods with exception handling constructs (and its percentage with respect to the total).

The smallest OSS analyzed (JUnit) has 15,502 bytecode instructions originated from 3369 lines of code, which means 4.6 bytecode instructions per line of code. In terms of the number of classes, JUnit has 239 classes and 48 out of 239 (20%) have exception handling constructions, at method level. JUnit has 1260 methods and 72 (5.71%) have exception handlers. On average, one may see that, at method level, the use of exception handlers is almost equivalent – varying from 4.61% to 11.72% of the total number of methods. This low percentage is similar to the data obtained by Sinha and Harrold [Sinha and Harrold 1998] for a different set of programs.

Table 2 – Complexity of OSS projects

OSS	Size	LOC	Number of Classes	Number of Classes with Exception Handlers	Number of Methods	Methods with Exception Handlers
HSQLDB	277,533	63,592	515	200 (38.83%)	8,318	683 (8.21%)
HTTPUnit	43,229	9,874	368	43 (11.68%)	3,829	121 (3.16)
JasperReports	254312	60,180	1,529	334 (21.84%)	1,4791	682 (4.61)
JMeter	161,385	38,693	773	285 (36.87%)	7,471	625 (8.37%)
JUnit	15,502	3,369	239	48 (20.08%)	1,260	72 (5.71%)

Log4J	3,6246	9,006	284	74 (26.06%)	1,919	225 (11.72%)
PMD	133,727	28,483	817	73 (9%)	5,901	374 (8.94%)
Velocity	45,569	10,191	269	80 (29.74%)	2,147	210 (9.78%)

After performing product static analysis, we started the dynamic analysis via JaBUTi. In JaBUTi's case, the dynamic analysis demands the instrumentation of each method of the SUT. The instrumentation is performed at bytecode level by using the JaBUTi instrumenter and, after that, we executed the available test set against this instrumented version, so that dynamic trace information could be collected and confronted with the structural testing criteria implemented by JaBUTi. , , and show the collected data.

and show the coverage after the execution of all available test sets developed by the OSS community for each program, considering the exception-independent and the exception-dependent testing criteria, respectively. For instance, the HTTPUnit test set was the one which determined the highest coverage with respect to all testing criteria. For All-Nodes-ei, the test set covered 6,889 out of 8536 required elements, 80.71% of coverage. As for the other testing criteria with higher complexity, the coverage percentages of the required elements were 78.53%, 76.70%, and 74.43%, respectively, for all-Edges-ei, All-Uses-ei, and All-Potential-Uses-ei. These coverage levels are compatible to the ones expected for CSS projects.

JUnit and Velocity presented structural testing coverage varying from 50% to 65% for all-Nodes-ei criterion, which is below the expected level for CSS projects, but can be considered a better result in comparison to HSQLDB, JasperReports, Log4J, and PMD. These latter projects have a coverage level below 40%, allowing us to infer that many codes are only executed by the users and that their test cases are probably not integrated in the official test set. In case of HSQLDB, the percentage of coverage of the All-Nodes-ei criterion is 19.73%, which means that more than 80% of source code is not executed by any official test case in the test set.

Table 3 – Requirement coverage: exception-independent criteria

Criterion	**All-Nodes-ei**	**All-Edges-ei**	**All-Uses-ei**	**All-Pot-Uses-ei**
OSS	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**
HSQLDB	8,029/40,703 (19.73%)	7,476/45,098 (16.58%)	19,720/126,246 (15.62%)	67,847/458,843 (14.79%)
HTTPUnit	6,889/8,536 (80.71%)	5,118/6,517 (78.53%)	1,0581/13,796 (76.70%)	23,156/31,110 (74.43)%
JasperReports	11,820/38,901 (30.38%)	9,619/37,469 (25.67%)	20,601/83,311 (24.73%)	74,118/293,119 (25.29%)
JMeter	7,845/20,462 (38.34%)	5,461/19,317 (28.27%)	10,935/41,180 (26.55%)	33,615/130,547 (25.75%)
JUnit	1,290/2,614 (49.35%)	844/1,873 (45.06%)	1,384/3,376 (41.00%)	3,373/7,936 (42.50%)

Log4J	2,043/5,602 (36.47%)	1,849/5,445 (33.96%)	4,071/10,759 (37.84%)	9,898/27,665 (35.78%)
PMD	7,938/21,184 (37.47%)	6,858/23,249 (29.50%)	13,331/57,552 (23.16%)	38,404/252,261 (15.22%)
Velocity	4,764/7,361 (64.72%)	4,587/7,929 (57.85%)	9,167/15,415 (59.47%)	25,555/46,499 (54.96%)

shows the obtained coverage with respect to the exception-dependent criteria, i.e., those criteria which demand an exception to be raised for covering the testing requirements. Considering the most basic structural testing criterion (All-Nodes-ed), the highest coverage was determined by the test set of the Velocity project, which executed 226 out of 1,006 testing requirements (22.47%). This is a low coverage and additional test sets should be developed at least to confirm that most of the exception handling construction in the program could be executed at least once.

Table 4 – Requirement coverage: exception-dependent criteria

Criterion	**All-Nodes-ed**	**All-Edges-ed**	**All-Uses-ed**	**All-Pot-Uses-ed**
OSS	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**
HSQLDB	141/1,942 (7.26%)	49/6,513 (0.75%)	256/2,750 (9.31%)	3,591/38,032 (9.44%)
HTTPUnit	37/221 (16.74%)	24/397 (6.05%)	34/161 (21.12%)	185/1,022 (18.10%)
JasperReports	5/1,493 (0.33%)	3/3,280 (0.09%)	3/1,294 (0.23%)	24/12,619 (0.19%)
JMeter	51/1,541 (3.31%)	39/4,863 (0.80%)	52/2,093 (2.48%)	276/15,301 (1.80%)
JUnit	12/156 (7.69%)	9/184 (4.89%)	13/183 (7.10%)	29/632 (4.59%)
Log4J	33/581 (5.68%)	10/1,070 (0.93%)	48/615 (7.80%)	194/3,954 (4.91%)
PMD	325/2,039 (15.94%)	121/3,814 (3.17%)	388/3,590 (10.81%)	1,689/20,285 (8.33%)
Velocity	226/1,006 (22.47%)	108/2,399 (4.50%)	356/1,769 (20.12%)	1,907/10,939 (17.43%)

In we present more detailed information about the total number of methods with exception handlers, the total number of testing requirements generated by the All-Nodes-ed criterion, the average number of requirements per method, the number of methods which do not have exception handler construction executed by any test case, and the total coverage obtained for such a criterion. As shows, there is a high percentage of methods with zero coverage against any exception-dependent criterion. Five out of eight programs have no test case to

execute their exception handling code for 90% or more of their methods. Two programs do not test the exception handling code for 80% or more; and only one program leaves untested 72% of their methods with exceptions. The best program is Velocity, for which the current test set is able to exercise 58 (28%) out of 210 methods with exception handlers, but still 72% of the methods are not executed by any test case.

Another point that might be inferred from is that the exception handlers have normally few nodes on average, i.e. they are not so complex in terms of logical structure. By analyzing such products, it is possible to observe that many exception handlers have empty `catch` blocks, just avoiding the exception propagation but with no corrective action associated with it. The most complex exception handlers are found in Velocity, which has on average 11.42 requirements per method, followed by PMD with 5.45 requirements per method, considering the All-Nodes-ed criterion.

These numbers show that all of the projects analyzed reveal a low level of code coverage for codes related to exception handling structures. This is disturbing because it reveals the lack of concern from OSS communities on constructing a reference test set for their products. Although it is possible to have a high quality software product using other activities for quality assurance, like formal review and inspection, testing is important to show the behavior of the product during its execution and a lower level of code coverage means that parts of the product are not being executed by the test suite.

Table 5 – Exception handlers data at method level: All-Nodes-ed criterion

OSS	Number of methods	Number of requirements	Average	Number of methods with no coverage	Total coverage (%)
HSQLDB	683	1,942	2.84	669 (97.95%)	7.26
HTTPUnit	121	221	1.83	101 (83.47%)	16.74
JasperReports	682	1493	2.19	679 (99.56%)	0.33
JMeter	625	1,541	2.47	595 (95.20)%	3.31
JUnit	63	156	2.48	57 (90.48%)	7.69
Log4J	210	1006	4.79	215 (95.56%)	5.68
PMD	374	2,039	5.45	299 (79.95%)	15.94
Velocity	210	2399	11.42	152 (72.38%)	22.27

For exception handling criteria the situation is even worse. Although the complexity of exception handlers is not high – as shown by the number of

testing requirements – the coverage of such testing requirements is very low. Many of the methods with this kind of code are not even executed once.

In addition, there is no indication of test cases specifically designed to address exception handling. Even if the adopted policy is not to execute exception handlers because they may be difficult to reach, the approach utilized in this assessment reveals which requirements could be neglected and which should be covered. An interesting approach for OSS communities and software developers in general would be the release of two test sets: one related to mainstream or happy path code and one to cover exception handling code.

presents the summary of coverage regarding ei and ed sets. One should consider the data of this table when accessing coverage data regarding All-Nodes, All-Edges, and All-Uses, and All-Potential-Uses. The data shows a modest coverage for HSQLDB, JasperReports, JMeter, Log4J, and PMD is modest (below 40%) while JUnit and Velocity coverage data is around 50%. In comparison to the recommended code coverage (70-80%), these OSS projects needs to improve their test suites to achieve the CSS recommended coverage level. The notable exception is HTTPUnit whose coverage level is right in the recommended level, between 70-80%.

Table 6 – Requirement coverage: exception-independent and exception-dependent criteria

Criterion	All-Nodes	All-Edges	All-Uses	All-Pot-Uses
OSS	Coverage (%)	Coverage (%)	Coverage (%)	Coverage (%)
HSQLDB	8,170/42,645 (19.16%)	7,525/51,611 (14.58%)	19,976/128,996 (15.49%)	71,438/496,875 (14.38%)
HTTPUnit	6,926/8,757 (79.09%)	5,142/6,914 (74.37%)	10,615/13,957 (76.06%)	23,341/32,132 (72.64%)
JasperReports	11,825/40,394 (29.27%)	9,622/40,749 (23.61%)	20,604/84,605 (24.35%)	74,142/305,738 (24.25%)
JMeter	7,896/22,003 (35.89%)	5,500/24,180 (22.75%)	10,987/43273 (25.39%)	33,891/145,848 (23.24%)
JUnit	1,322/2,780 (47.55%)	865/2,076 (41.67%)	1,424/3,583 (39.74%)	3,474/8,575 (40.51%)
Log4J	2,076/6,183 (33.58%)	18,59/6,515 (28.53%)	4,119/11,374 (36.21%)	10,092/31,619 (31.92%)
PMD	8,263/23,223 (35.58%)	6,979/27,063 (25.79%)	13,719/61,142 (22.44%)	40,093/272,546 (14.71%)
Velocity	4,990/8,367 (59.64%)	4,695/10,328 (45.46%)	9,523/17,184 (55.42%)	27,462/57,438 (47.81%)

2.4. Dynamic measures for size and coupling

In this section, different coverage measures are used to compute dynamic measures for size and coupling. The dynamic size measures are similar to the control-flow testing criterion all-Nodes (blocks, instructions, and SLOC

measures). Other size metrics are related to coverage of methods and types in Java programs. We are interested in collecting dynamic data to show the gap between dynamic and static measures (e.g., total number of blocks, instructions, SLOC, methods, types).

Moreover, we aim at preliminary investigating whether AOP and coverage criteria are powerful enough to collect dynamic measures of size and coupling.

Coverage Criteria for Dynamic Size Measures

Blocks: The number of blocks (i.e., the sequence of bytecode instructions without any jumps or jump targets) exercised by the test suite. A block is considered as exercised when its last instruction has been executed. A module in this case is statically modelled as a block control-flow, in which bytecode blocks are represented by the nodes of the graph (i.e., the elements of the system) and control-flow transfers between blocks are represented by the arcs of the graph (i.e., the relationships of the system).

Instructions: The number of bytecode instructions exercised. A module in this case is statically modelled as a control-flow graph, in which bytecode instructions are represented by the nodes of the graph and control-flow transfers between bytecode instructions are represented by the arcs of the graph.

SLOC: The number of Java source lines of code exercised. A module in this case is statically modelled as a control-flow graph, in which source lines of code are represented by the nodes of the graph and control-flow transfers between source lines of code are represented by the arcs of the graph.

Methods: The number of distinct methods exercised. A method is considered as exercised if at least one statement of the method has been executed. A module is statically modelled as a sheer sequence of methods declared in a class, so methods are represented by the nodes of the graph. The node representing a method is linked to the next node (which represents the next method in the sequence) by an arc.

Types: The number of distinct Java type exercised. A Java type is considered as exercised if it has been loaded and initialized.

Coverage Criteria for Dynamic Coupling Measures

We collected three dynamic coupling measures, defined in [Arisholm et al.

2004].

Dynamic messages: The count, within a runtime session, of the total number of distinct messages sent from one object of a class to other objects. Two messages are not distinct if their source and target classes, the method invoked in the target class, and the statement from which it is invoked in the source class are the same. Here, one may model a class as a module and each method as an element. Thus, a method invocation is represented as a relationship between an element in one class and an element in another class.

Distinct method invocations: The count, within a runtime session, of the total number of distinct methods invoked by each method in each object.

Distinct classes: The count the distinct number of classes that a method uses within a runtime session.

Application example of dynamic measures

We present the results of a case study conducted to provide a first empirical validation of the dynamic measures discussed above. First, we present the methodology we follow, the environmental setup and the objectives of this experimentation. Second, we provide and we discuss quantitative results about size and coupling dynamic metrics.

Objectives and Methodology

We selected an open-source Java project called RealEstate to evaluate dynamic size and coupling aspects. RealEstate is a software application created at North Carolina State University that reproduces the Monopoly game [NCSU, 2009]. RealEstate is released within a test suite that contains a set of unit and acceptance test cases. The considered RealEstate release consists of 2723 source lines of code (SLOC) in 4 source packages, 79 classes and 569 methods (after removing classes related to test cases and methods used to profile the source code of the application).

We focused on collecting different types of data referring to the dynamic size and the coupling of the application. As for collecting dynamic size measures, we used the EclEmma Eclipse plugin [EclEmma, 2010]. EclEmma is a free Java code coverage tool for Eclipse that does not require modifying the RealEstate application. We used the test cases provided with the RealEstate source code to exercise the system, and by means of EclEmma we are able to compute at run-time the measures we discussed above.

Following our previous experience and results, as for collecting dynamic coupling measures, we profiled the RealEstate application by means of Aspect-

Oriented Programming (AOP) and we exercised the system by means of the RealEstate test suite. We used the AJDT Eclipse plugin (based on the AspectJ runtime library) [AJDT, 2010] to define an aspect (i.e., a stand-alone module that contains cross-cutting concerns), a pointcut (i.e., a set of join points that are able to capture well-defined moments in the execution of a program, like method call, object instantiation, or variable access), and an advice (i.e., the code to run before, after, or around the specified join point) able to trace all the calling classes and methods, the called classes and methods, and the statements from which calls originate. The pseudo code of the most relevant part of the defined aspect looks like as follows:

```
public aspect DynamicCouplingAspect {
    pointcut executionTree():
        within(TestClass_a) ||
        ... ...
        within(TestClass_n) &&
        call (* *(..));
    Object around(): executionTree() {
        print(thisJointPointStaticPart.getSignature);
        print(thisJointPointStaticPart.getDeclaringType);
        ... ...
    }
    ... ...
}
```

Experimental Results

This subsection summarizes the main results about dynamic size and coupling measures. shows the results of the measures related to size aspects discussed in the previous subsection. Column <Target Element> specifies the code element that is monitored by EclEmma when executing the RealEstate test suite; Column <Static Measure> indicates the total number of occurrences we statically detected into the system for the target element; Column <Dynamic Measure> reports the total number of occurrences we dynamically detected into the system for the target element when executing the test suite; Column <Ratio> specifies the percentage of exercised elements at run-time over the total number of static elements.

Table 7 — Dynamic Size Measures Results for the RealEstate Java application

Target Element	Static Measure	Dynamic Measure	Ratio
Blocks	901	387	43.0%
Instructions	10411	4775	45.9%
SLOC	2723	1316	48.3%

Methods	569	247	43.4%
Types	111	57	51.4%

As shown in , it is clear how dynamic measures are different from static measures. Of course, these results are strongly related to the quality of the test suite that can or cannot stress specific aspects of the system. shows the results of the three coupling measures previously discussed. We randomly selected different sets of test cases and we executed the system with these different scenarios to understand whether a linear reduction on the number of test cases implies a linear reduction of the dynamic coupling.

In Figure 1, Figure 2, Figure 3, and Figure 4 we show the trend of the collected data about the three dynamic coupling measures across the variation in the number of test cases that stimulate the system. As shown in these figures, the coupling linearly grows with the number of the inputs (i.e., the number of the test cases) that stimulate the system.

This example confirmed that AOP is a valid approach for detecting dynamic measures of a software system: it requires a limited effort for defining aspects and pointcuts able to trace data about dynamic coupling measures, it does not impact the system performance, and it favours the separation of concerns (i.e., the source code of the system is separated from the source code of the aspects).

Of course, additional experiments are needed to understand whether AOP is able to successfully monitor more complex dynamic measures than the one presented in this work. Additionally, the linear correlation between dynamic coupling and number of test cases observed in RealEstate is verified in other OSS projects.

Table 8 -- Measures results for the RealEstate Java application

Execution Scenario	#of Dynamic Messages	#of Distinct Methods	#of Distinct Classes
Scenario 1	1324	135	48
Scenario 2	557	99	30
Scenario 3	614	62	26
Scenario 4	674	87	27
Scenario 5	288	56	16
Scenario 6	217	53	21
Scenario 7	162	31	10
Scenario 8	71	27	8

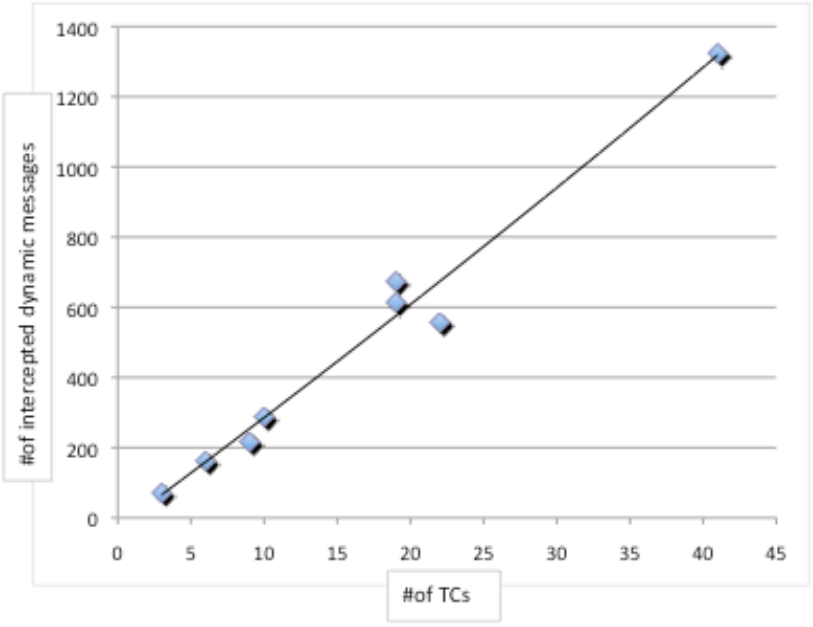

Figure 1 -- Number of intercepted dynamic messages x number of test cases.

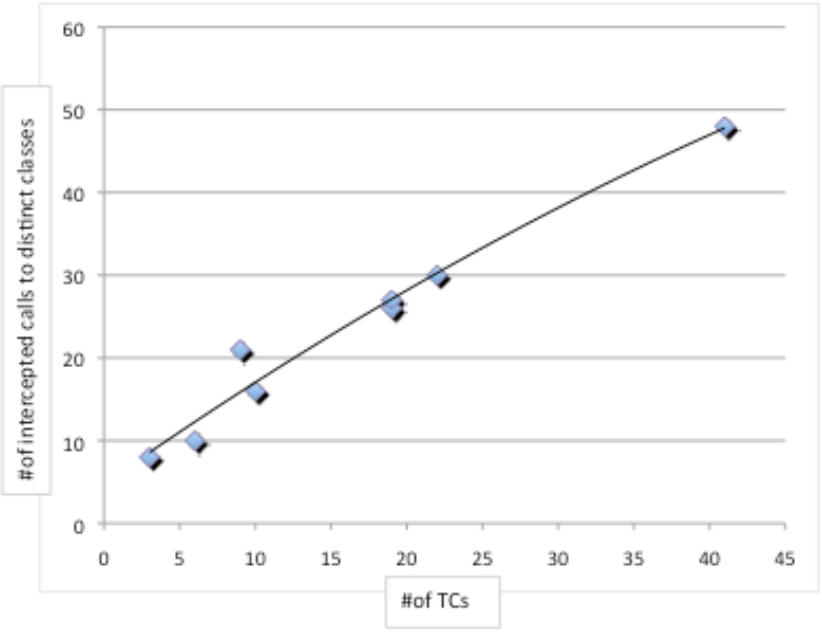

Figure 2 -- Number of intercepted dynamic calls to distinct classes x number of test cases.

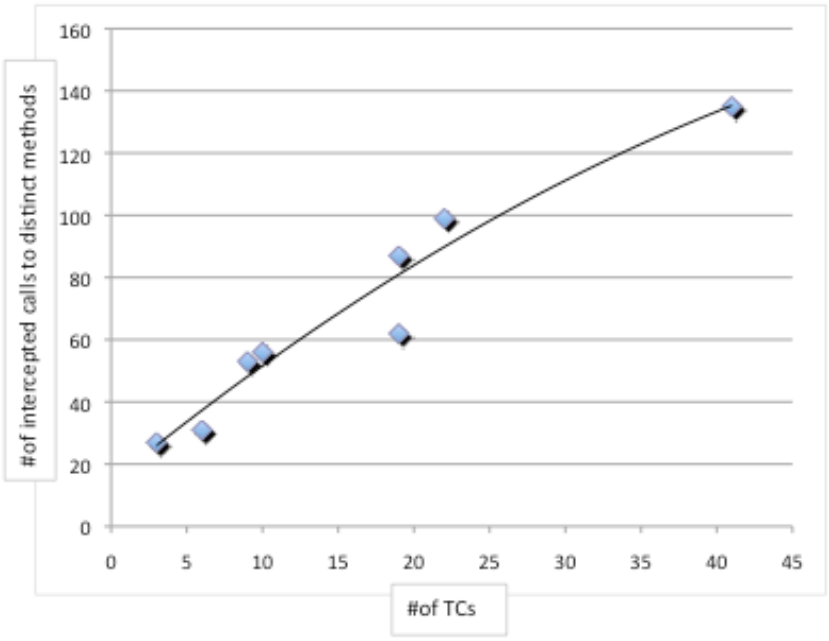

Figure 3 -- Number of intercepted calls to distinct methods x number of test cases.

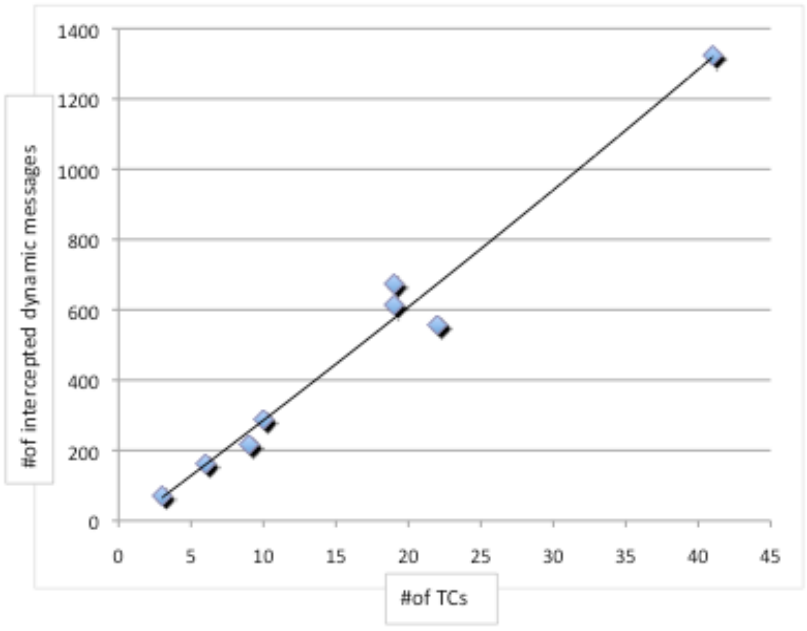

Figure 4 -- Number of intercepted dynamic messages x number of test cases.

2.5. Related work

Code coverage has been investigated as a measure to assess the quality of a test set. Experiments to assess the *effectiveness* of test sets of different coverage levels have been conducted [Frankl and Weiss 1993] (Frankl and

Iakounenko 1998) [Hutchins, et al. 1994]. By effectiveness of a test set one should understand its likelihood in revealing errors of a program. In general these experiments utilize a set of programs that have a pool of test cases so that it is possible to develop a great number of different test sets with identical level of code coverage. The idea is to obtain a number of similar test sets in such a way that relevant statistical analysis can be produced.

[Ho, Elbaum and Rothermel 2005] developed an infrastructure to support the design and conduction of experiments to evaluate testing and debugging techniques. The authors made available several programs, with their respective test pools and erroneous versions, to facilitated experiments to assess the effectiveness and efficiency of testing and debugging techniques. The majority of programs are OSS and one of them is JMeter.

The data presented in this paper and in the previous works differs on purpose. Our data aims at assessing the thoroughness of the testing process that generated the OSS test suites. On the other hand, the previous works' goal is to assess the effectiveness of the structural testing techniques.

2.6. Final remarks

We presented experimental data collected by JaBUTi in a set of eight OSS projects. The experiment intended to assess the adequacy of pre-existent test sets against a set of structural testing criteria.

Our observations reveal that in general the coverage with respect to structural testing criteria needs improvement. When evaluating the quality of the pre-existent test sets against the exception-independent criteria, we have found a single project with coverage at the 70%-80% level for the All-Nodes-ei criterion. We have found two projects with coverage in the 50%-65% range and four projects with coverage below 40%, which is generally regarded as a low level of coverage and an indicator that the test set should be improved

The coverage regarding exception-dependent criteria is more modest. For instance, the maximum coverage of the All-Nodes-ed criterion was below 22.5%, which shows that, in general, there is no concern for the development of test cases to exercise exceptional conditions in the project. Moreover, many exception constructions have empty `catch` blocks, which reveal that the exception handler, though present, is used only to avoid the spread of the exception, not to recover from an erroneous condition.

The overall data, considering exception-independent and exception-dependent coverage, reveals the need of improvement of code assessment in the OSS projects. A single project obtained coverage above 70% for all-Nodes and all-Edges. Both criteria are considered basic structural testing with a minimal coverage of 70% recommended for CSS.

Similar control-flow coverage metrics were collected for a small program with a different tool and using AOP resources. The results are along the same lines: the number of blocks, instructions, SLOC, methods, classes, and types

exercised varied from 43% to 51.4%. Dynamic coupling metrics were successfully collected using AOP resources. The dynamic coupling metrics grew linearly as the number of test cases augmented. This observation deserves further examination considering other OSS projects. Moreover, AOP greatly facilitated the collection of such metrics; however, AOP scalability should still be verified for bigger OSS projects.

Concerning the testing criteria coverage, these are rather unexpected results since the OSS projects investigated are well-regarded and commonly used in many industrial settings. One possible hypothesis for this behaviour is that many tests are *not* added in the test suites. A developer may fix or add a new feature, creates a test to verify it, but does not add it to the test suite. Furthermore, much of the testing being carried out in the OSS context is expected to be performed by final users. They might explore the code of the OSS product but their informal tests are not registered in test suites.

This characteristic of the OSS development highlights some shortcomings of an OSS product when compared to a CSS product. Poor test suites undermines future code refactoring and regression testing, implying difficulties for performing maintenance of the code. Moreover, an industrial user will hardly change a CSS product by an OSS product without being able to assess the OSS product trustworthiness in comparison to a CSS counterpart. In this sense ways to assess the testing process of OSS products are needed. In the next section we introduce the OSS-TMM – Open-source Software Testing Maturity Model – whose objective is to assess the maturity of testing process which takes place in the OSS realm.

Although a 100% of code coverage is not a guarantee of a high quality software product, higher the coverage, higher the confidence the product behaviour is correct or, at least, the executed code is necessary to provide the product functionality. Besides structural testing, other quality assurance activities, such as inspection and formal review, may also be used to maximize the defect detection rate earlier as possible.

The coverage metrics presented in this section may be used in MOSST – Model of Open Source Software Trustworthiness – which aims at assessing the trustworthiness of OSS projects.

3. OSS-TMM – Open source software testing Maturity Model

In this section, we define a Maturity Model for the testing process of OSS systems (OSS-TMM: Open Source Software Testing Maturity Model). We introduced the idea of OSS-TMM in [Tosi, Taibi and Morasca 2009]. OSS-TMM aims at improving the quality and the trustworthiness perception of OSS products by supporting the planning, monitoring, and execution of testing activities. Unlike existing certification models [Burnstein, Suwanassart and Carlson 1996], [Herbsleb, et al. 1997], [Emam 1997] which have been defined with CSS characteristics in mind, our model takes into account the specific issues that characterize and distinguish OSS systems from CSS systems, how these differences influence the testing process, and the relationships between these issues and the testing techniques applicable to OSS systems.

Specifically, OSS-TMM provides guidelines for improving an OSS testing process by recommending the checks that need to be done to retrieve as many failures as possible by means of specific testing activities and test suites. OSS-TMM is based on a list of common testing issues that characterize OSS products, which we identified as the result of experience we acquired by both evaluating 32 well-known OSS projects (such as the Linux OS, the web server Apache, and the GCC compiler) and analyzing the literature that focuses on OSS products. To provide evidence for its usefulness, we have applied our maturity model to the BusyBox and the Apache HTTP OSS products, to show how our model can actually improve testing process quality in real-life projects. Additionally, we tried to answer the question: Does a high maturity of the testing process directly mean a high quality of the OSS product, and does a low maturity level directly mean a low quality of the product? To answer this question, we correlated the maturity level of a representative set of OSS products with their bug rate.

In what follows, we discuss the motivations, goals and details of our approach, describe our experience with BusyBox, Apache HTTP and TPTP, and report on the maturity level of four additional OSS projects.

3.1. Towards a maturity model for Open Source Software

Our experience in the context of OSS projects suggests that OSS communities do not usually view software testing as a primary software development activity. Also, most OSS projects do not integrate testing activities into their development process. In a survey, we asked 151 OSS users (developers, contributors, final users, etc.) and stakeholders to rate the importance of a number of factors that they take into account during the adoption of OSS components and products. The complete survey can be found in [QualiPSo1 2009]. Unexpectedly, interviewees on average answered that the factor "existence of benchmarks / test suites that witness for the quality of OSS" takes

a low importance. This may be a result of the fact that benchmarks and complete test suites are hardly ever available for OSS, more than the fact that benchmarks and test suites might not be important. So, OSS end users, integrators, stakeholders, etc. do not use benchmarks and test suites simply because they often do not exist. We also analyzed the web portal of 32 well-known OSS products and we discovered that in the web-portals only 6% of the products show the availability of test suites; 19% of the products provides performance benchmarks; 3% show the usage testing framework to support testing activities; 0% provides results about test suites executions; 41% provides internal (or external) reports about benchmarks executions. The complete list of projects can be found in [QualiPSo2 2009]. These somewhat discouraging data are in contrast with the trend followed by CSS products where software testing is considered as one of the most important activities of the development process.

The final goal of our work is to help OSS developers test their products, by defining a Maturity Model that can be used by companies, developers, and final users to assess and improve the testing process of the OSS product under consideration. To support this goal, we identify:

- A set of maturity levels (MLs) that reflect the evolution of the OSS testing process.
- The set of issues that characterize OSS systems and that point out the differences between OSS and CSS products, and also a set of guidelines that will be used to identify the testing techniques that best fit the characteristics of OSS products. These two sets are presented in Appendix A as a checklist of issues that can be used by companies and private developers/contributors to identify the peculiarities of their OSS products, to discover the level of compliance of the target OSS product with the typical OSS characteristics, and to define the best testing process for the target product.
- A step-by-step methodology that companies, private developers/contributors, and final users can follow to assess the maturity level of the testing process available for the target OSS product.

3.2. Maturity level

In compliance with existing certification and maturity models, we identified four maturity levels that reflect the evolution of the testing process from one that is unstructured and undefined (Level 1) to one that is well planned, monitored and optimized (Level 4). Refer to Section 3.6Errore: sorgente del riferimento non trovata for more details about other certification and maturity models.

Unlike in CMM and TMM [Herbsleb, et al. 1997], [Burnstein, Suwanassart and Carlson 1996] our levels are not defined and structured as sets of predetermined maturity characteristics and goals, but they depend on the specific characteristics of the product under evaluation. Burnstein et al.

[Burnstein, Suwanassart and Carlson 1996] define five maturity levels of a testing process starting from Level 1 in which the testing process is initial and not distinguishable from debugging, to Level 5 in which the testing process has a set of defined testing policies, a test life cycle, a test planning process, a test group, a test process improvement group, a set of test-related metrics, appropriate tools and equipments, controlling and tracking mechanisms, and finally a product quality control. Such a model is unsuitable in the OSS scenario where the testing process strongly depends on the inherent issues and characteristics of the target product. In our experiments, we quickly applied Burnstein's Testing Maturity Model (TMM) to the 32 OSS products of our experiments, and we discovered that none of these products obtains a maturity level greater than 3, and the vast majority of products fall into the first and second level.

Hence, we identified four maturity levels with less stringent requirements than TMM, which are dynamically computed for each product by applying our OSS-TMM. Next, we describe these four maturity levels. At any rate, the range of values reported in the following list can be refined over time to normalize the values based on the results obtained from a more extensive evaluation of our approach. In the formulas in the following list, *BTP* and *ATP* represent the sets of activities of the Best and Available Testing Processes, respectively. By "best testing process," we mean the most mature process it is theoretically possible to achieve with reference to the inherent characteristics of the product. A testing process is mature if it has been structured for completeness (i.e., appropriate testing activities are planned to detect each important class of faults that depends on the application domain, the organizations and the technologies employed), timeliness (i.e. faults are detected as soon as possible), and cost effectiveness (i.e., testing activities are chosen depending on their cost as well as their effectiveness). Of course, the definition of the *BTP* is not fully objective due to the huge number of testing techniques and practices that are potentially useful. Our intent is to suggest a set of representative testing activities and technologies, and not a rigid model, due to the rapid evolution of the field, especially in the OSS world.

- Level 1: The activities performed by the ATP cover the activities suggested by BTP with a degree that is lower than 25%. As a formula: $|ATP \cap BTP| < 25\% |BTP|$;
- Level 2: The activities performed by the ATP cover the activities suggested by the BTP with a degree that is in the range 25%-50. As a formula: $25\% |BTP| <= |ATP \cap BTP| < 50\% |BTP|$;
- Level 3: The activities performed by the ATP cover the activities suggested by the BTP with a degree that is in the range 50%-75%. As a formula: $50\% |BTP| <= |ATP \cap BTP| < 75\% |BTP|$;
- Level 4: The activities performed by the ATP cover the activities suggested by the BTP with a degree that is in the range 75%-100%. As a formula: $75\% |BTP| < |ATP \cap BTP| <= 100\% |BTP|$.

3.3. OSS issues

The inherent issues (and sub-issues) that characterize OSS products are related to the following five macro categories:

- (I1) visibility of an OSS product (i.e., the availability of the source code and its internal structure)
- (I2) system analysis and product design activities
- (I3) development process
- (I4) system growth and community creativity
- (I5) documentation and dissemination aspects.

In this section, we informally discuss each macro category, the sub-issues that belong to each category, how OSS products differ from CSS systems, and the connections between issues and testing methodologies. In Appendix A we schematize these categories.

3.3.1. I1 - Visibility

Contrary to CSS, OSS is freely distributed and the source code is open and transparent to both developers and end-users under specific license policies. The full visibility of the internal workings of the system (i.e., the logic and the structure of the code) provides developers and users with the opportunity to exercise and test the complete behaviour of the system. Visibility facilitates the applicability of all the software testing techniques that fall into the "white box" testing category [Pezzè and Young 2007] and that address: (1) *unit testing*, used to detect defects in each component before it is released and integrated with other code; (2) *integration testing*, used to check for defects during the integration process of components; (3) *regression testing*, used to selectively retest the system to check whether a modification of the code has caused unintended effects. *Path testing*, *data flow testing*, and *code inspection* [Pezzè and Young 2007] are examples of verification techniques that may address the requirements imposed by OSS systems.

Visibility also implies the possibility to have log files or execution traces available to the OSS community. This can suggest the possibility of using techniques of *dynamic analysis* to analyze log files, compute behavioural properties that are true for the collected data, and then check those properties against future executions to detect misbehaviours [Ernst, et al. 2001].

Another important aspect to take into account when testing OSS systems is related to security. It is obvious that the availability of the source code potentially increases the vulnerability of the system, thus making it important to carry out a serious campaign of security testing at each layer if necessary. Detecting software security vulnerabilities is not a trivial task and requires *ad hoc* verification and testing techniques able to discover security issues (e.g.,

secure code review, *symbolic execution*, *risk-based security tests*, *penetration tests*, *dependencies tests* [Howard 2006], [Dannenberg and Ernst 1982], [Arkin, Stender and McGraw 2005]). In the context of OSS products, the scripts' source code of the test cases is also available, thus implying the possibility that malicious developers may manipulate the script.

3.3.2. I2 - System Analysis and Product Design Activities

System analysis and product design are usually not well planned activities in OSS. This is due to the short-term and non-commercial vision that characterizes some OSS projects, many of which were started to solve a user's particular problem without a long-term vision and a real perception of the innovation and evolution degree that the project could have in the future (this is the case of Linux, Perl, and the World Wide Web). The evolution of each release of OSS projects only depends on the unpredictable spread and diffusion that the project may have, and it is directly related to the attraction that the project produces in the community over time, thus making it impossible to pre-plan the design of each release of the system.

Moreover, OSS is often characterized by an unstructured environment, where tasks are not assigned, because OSS developers are volunteers who mostly do what they want to do. In CSS projects, team members have assigned work; in OSS projects, team members choose work. Due to this freedom, activities that are viewed as nuisances such as project plans definition, system design evaluation, and requirements analysis may not be adequately performed in the OSS community.

As a consequence, system requirements are not always defined in advance by skilled analysts, but are discussed over time by the developers. Risks are not formally assessed in advance, nor are they monitored or formally managed during the project life-cycle. Whenever system analysis and system models are unavailable, all the testing techniques based on system specifications (such as *model-based testing* [Pretschner, et al. 2005], *category partition* [Ostrand and Balcer 1988], etc.) are not well applicable to the OSS product, and a pre-planned testing plan is infeasible.

While goal, risks, and models are not always provided, performance requirements and hardware/software requirements are often highlighted. This encourages the applicability of testing techniques (such as *load testing*, *stress testing*, and *endurance testing* [Pezzè and Young 2007]) that test the performance of the whole system and combinatorial testing techniques to check all the possible pair wise combinations of hardware and software platforms supported by the product. If external or third-party libraries and plugins are required, versioning compatibility checks should be performed to avoid integration problems. Whenever coding standards and coding conventions are required, style check and inspection techniques should be applied to check the compliance of the released code with the required conventions. Moreover, when the product requires a graphical user interface (GUI), usability tests should be

planned, and “capture & replay” tools (such as Jacareto [http://jacareto.sourceforge.net]) could be used to automate the re-run of specific program executions.

3.3.3. I3 - Development Process

OSS is developed in a collaborative and distributed way [Raymond 2001]. OSS systems are developed in a large-scale cooperative context, where different teams, private users, a passionate core of developers, and virtual communities create the “unstructured company" (E. S. Raymond called it “Bazaar" [Raymond 2001]) that will contribute to the project. Internet is the scaffolding and the desktop for these virtual software development organizations, and developers are coordinated by simple license policies without mechanisms of hierarchy and supervision as stringent as in CSS development. As a consequence, OSS developers hardly ever follow a development methodology that is as well defined as that followed by CSS developers.

In this scenario, classical development processes, such as the Waterfall model or the Spiral model, are often inapplicable. The development process of OSS products often resembles Agile or XP models in which a cycle of test design/execution is wrapped around each small-grain incremental development step. This makes it necessary to focus on testing techniques that are iterated during the whole development process of the OSS product. Continuous [Saff and Ernst 2004] and evolutionary testing [Santelices, et al. 2008] are new techniques that are available to this end.

Focusing on testing aspects, the issue I3 may favour a more rapid discovery and fix of defects than in CSS projects [Mockus, Fielding and Herbsleb 2000] since developers and end-users are unaware testers of the system [Raymond 2001]. When developers provide a new system feature, they test the coded functionality and also the entire system in their own sandbox environment, thus implicitly providing a set of test cases that are unique (because they depend on the characteristics of the sandbox environment). This is also true for end-users that install and use the latest version of the system, providing important usability feedback to the community. Moreover, in some OSS projects, patches and new functionalities are made available as soon as they are developed. In CSS systems, they are bundled into new scheduled releases thus slowing down the process of “customer testing." This suggests the idea of collaborative testing where developers and end-users should share testing knowledge with each other to allow the applicability of fault-based testing [Morell 1990].

However, unstructured teams have more difficulties in planning the testing activities, and more risk to introduce errors every time they release a program change because they do not know the impact that your change can have on the system. This requires the introduction of a strong regression testing activity to re-run previously executed tests and check whether previously fixed faults have re-emerged due to program changes.

Moreover, most of the developers are not skilled testers. Thus, they resort to

adopting an integrated development environment (IDE) that provides comprehensive facilities to computer programmers for software development; to exploiting all the testing facilities provided by the IDE (e.g., in ECLIPSE many plugins for testing are provided); to introducing into the development process automated testing platforms and frameworks (such as TPTP: the Eclipse Test & Performance Tools Platform Project [Eclipse 2009]) that help plan the testing process, automatically derive oracles and stubs (to reduce the effort in writing test cases), document the executed test cases (to simplify the process of reusing test cases), report test results in a well agreed format (to facilitate the interpretation of the results), and monitor the testing process.

3.3.4. I4 - System Growth and Community Creativity

OSS is characterized by a faster system growth [Mockus, Fielding and Herbsleb 2000] and more creativity than CSS [O'Reilly n.d.], which may lead to a more rapid satisfaction of customer needs. This is primarily due to the unstructured and informal organization of the communities. It is often believed that structure and rules “inhibit innovative thinkers and drive them to the fringes [O'Reilly n.d.]," while informality and freedom boost action and creativity. This implicitly requires the definition of architectures that are inherently modular and scalable to guarantee the extensibility of a system and its interoperability across different hardware and software platforms.

Focusing on testing aspects, I4 highlights the importance of regression testing activities to avoid bugs introduced by a lavish creativity. Up to date test cases should be made available to the community to facilitate the process of re-executing the whole test suite. Moreover, the iterative process of testing the single and integrated units, and then retesting the entire system behaviour should follow the rapid evolution of the system. Online built-in testing methodologies may simplify this “keep-in-touch activity" by means of automatic instrumentation and profiling of the code (via *aspects*, *probes*, and *monitors*) [Mao, Lu and Zhang 2007]. These techniques dynamically collect input-output and interaction data to facilitate the identification of functional and non-functional misbehaviour of the system under control. Moreover, these techniques could support the process of customer testing and continuous testing by simplifying the collection and execution of transparent test cases.

Proportionally to the size of the community and the vitality of the project, developers of OSS products should improve: testing automation, to simplify the generation of oracles and stubs; regression testing, to avoid conflicts that arise from program changes; the definition of acceptance tests, to avoid that some feature of the completed product is untested; the sharing of testing knowledge, to increase the reusability of test suites; the documentation of test-strategy/test-plan/test-design/tests-results, to simplify the monitoring of the testing process.

For example, in our experience [QualiPSo2 2009], very few products use available testing frameworks to support testing automation.

3.3.5. I5 - Documentation and Dissemination

Poorly structured documentation, user manuals, bug reports, and technical reports often characterize OSS. This is related to the fragmentation of human resources: contributors prefer to focus their effort on coding rather than writing documents. Moreover, the use of the network and of distributed resources fosters the dissemination of the project knowledge via unstructured channels (e.g., mailing lists, forums, and chat logs). Finally, the high modularity of the projects and the frequency of changes do not favour the comprehension of the global meaning of the entire project, thus fostering library-level documentation instead of system-level documentation. A solution for this challenge is not simple, for several reasons: for instance, it is not possible to either force contributors to write documentations or employ technical writers; good documentation requires skills that may not be always found in OSS developers.

While the unavailability of system documentation complicates acceptance / system testing, usability testing, installation testing, and the sharing of testing knowledge, the unavailability of testing documentation complicates the monitoring of the whole testing process. The use of test management tools can mitigate this problem, by simplifying the organization of the testing activities and by automating the generation of testing reports.

For example, in our experience [QualiPSo2 2009], only 1 product (out of 32) provides a complete documentation about its internal testing activities. Only JBoss [www.jboss.org] exposes a detailed and up-to-date documentation about testing plans, testing methodologies, test cases description, and test suite results.

3.4. OSS-TMM-based Process Assessment

The approach we propose for companies, private developers/contributors, and final users to assess the maturity level of the testing process of their products, is compliant with the ISO/IEC14598 standard [ISO1 2001], which gives guidance and requirements for evaluating software processes. OSS-TMM is based on four main steps (S) as indicated in Figure 5, with an additional (optional) step (S5):

S1: take into account the issues discussed in Section 3.3 and analyze the target OSS product with reference to these issues. For each issue (and sub-issue), verify the fulfilment degree of the product. To simplify this step, users can use the checklist to sequentially examine the target product;

S2: define the Best Testing Process (BTP) on the basis of the results of step S1. To simplify this step, users can apply the checklist (presented in Appendix A) to identify the proper testing activities;

S3: isolate and briefly analyze the currently Available Testing Process (ATP) of the product in order to list the properties and the already used testing techniques;

S4: verify the intersection degree between the activities of the testing process model derived in step S2 (*BTP*), with the ones analyzed in step S3 (*ATP*), and identify the maturity level (*ML*) of the testing process referring to the maturity levels identified in Section 3.2;

S5: final users can use the maturity level as an indicator that contributes to assessing the quality and the trustworthiness of the OSS product they are evaluating. Otherwise, they can use the maturity level to evaluate whether their ATP needs improvement. If this is the case, they can improve the testing process by following the recommendations and guidelines provided in BTP.

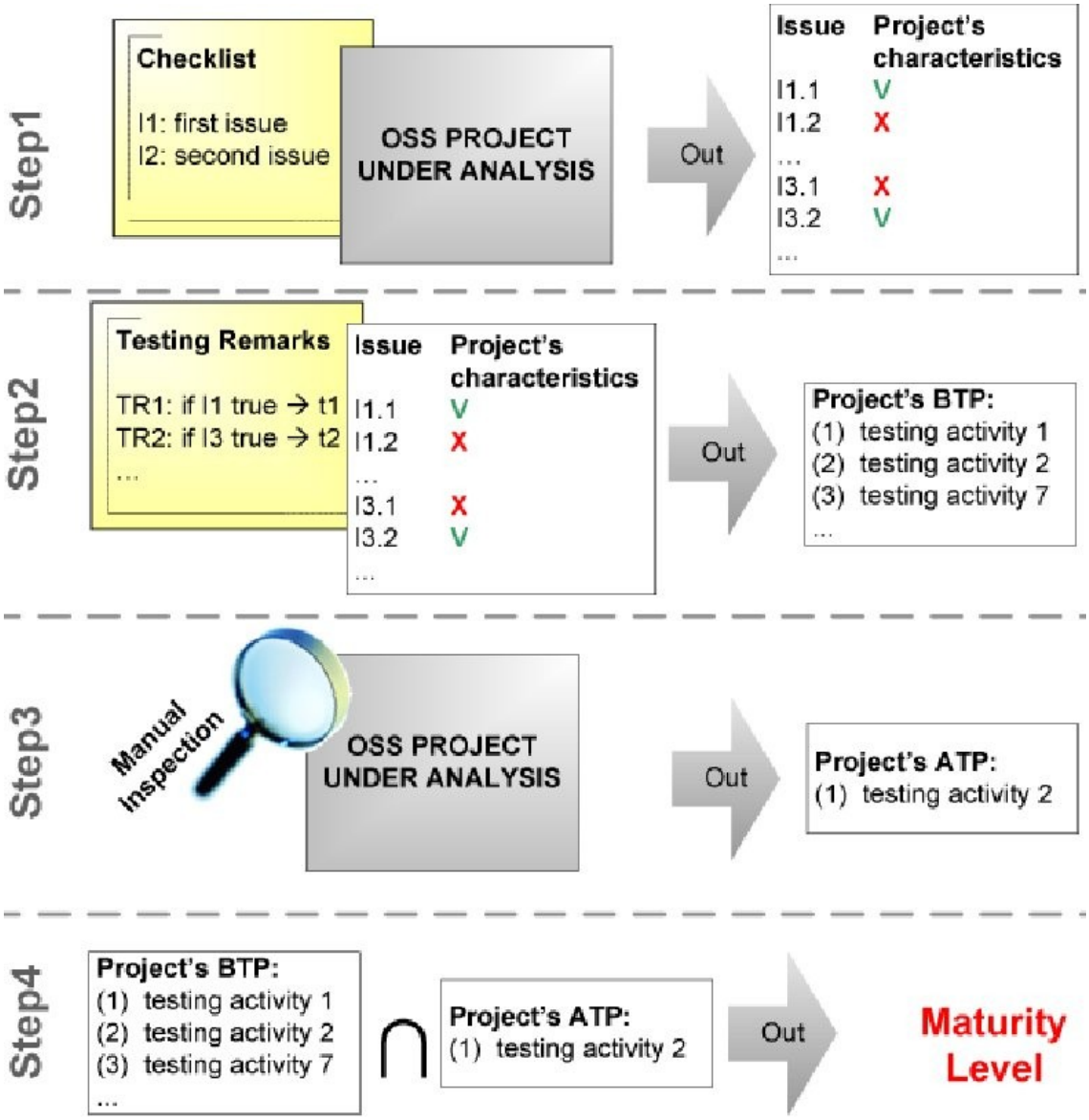

Figure 5 – OSS-TMM main steps.

3.5. Preliminary Results

We empirically examined the OSS-TMM with two case studies. The first one (BusyBox) with a focus on how a developer can improve the testing process of his product, while the second one (Apache HTTP) with a focus on how a final user can estimate the maturity of the product under evaluation. In this section, we further report the results obtained by the correlation of six OSS maturity levels with respect to their bug rate.

3.5.1. BusyBox evaluation

Here, we exemplify the methodology discussed in Section 3.2 with the BusyBox OSS project. We applied OSS-TMM to BusyBox from the developer point-of-view with the goals of 1) demonstrating the simple applicability of the OSS-TMM to derive the maturity level of the product; 2) verifying whether the suggestions provided by our model can actually improve BusyBox's testing process.

BusyBox [`www.busybox.net`] is an OSS project, developed in C, which has the typical properties of OSS projects. BusyBox combines tiny versions of common UNIX utilities into a single small executable, providing minimalist replacements for the utilities usually found in Linux environments.

In this case study, we sequentially applied all the steps of our methodology: (1) we analyzed BusyBox by scanning and answering each entry of the checklist; (2) we identified the *BTP* in relation to the actual characteristics of BusyBox; (3) we estimated the maturity level of the product; and (4) we redefined the BusyBox *ATP* by following the suggestions provided by the BTP.

Step 1: *Analysis of Issues*.

Table 9 summarizes the BusyBox characteristics derived from the analysis of the project through the checklist.

Table 9 – Step 1 outcome for BusyBox

Issue	BusyBox characteristic
I1.1	the whole project is managed via SVN
I1.2	the whole project is well structured in 28 folders
I1.3	information about releases are visible
I1.4	information about code revisions are visible
I1.5	sensible data are manipulated (e.g., username, pwd)
I1.6	all the scripts are open source to the community
I1.7	the popular license GPLv2 is used
I1.8	log files are not available
I2.1	the project plan/roadmap is unavailable
I2.2	the risk analysis is unavailable
I2.3	the requirements analysis is unavailable
I2.4	the goal analysis is unavailable
I2.5	UML diagrams are unavailable
I2.6	the standard "*Shell and Utilities OGB*" is used
I2.7	coding standards and conventions are not identified

I2.8	performance requirements are not meaningful
I2.9	BusyBox does not follow a specific architectural style
I2.10	BusyBox is designed without a GUI
I2.11	BusyBox does not integrate external libraries/plugins
I3.1	none specific development process is followed
I3.2	developers are unstructured
I3.3	a sandbox environment is not provided
I3.4	none specific IDE is used/recommended
I3.5	none specific testing platform is used/recommended
I3.6	the *Bugzilla* bug tracking system is integrated
I4.1	24110 revisions in total
I4.2	38 developers/contributors
I4.3	the analyzed release is: V1.14.0
I4.4	the number of open/fixed bugs is provided
I4.5	BusyBox is a vital project
I5.1	the system-level documentation is unavailable
I5.2	the library-level documentation is unavailable
I5.3	a simple features-level documentation is available
I5.4	a short user manual is available (README file)
I5.5	bug reports are available
I5.6	the code documentation is unavailable
I5.7	documents are also disseminated via a mailing list
I5.8	installation requirements are not documented
I5.9	test documentation is unavailable

Step 2: BTP derivation.

The data collected during the previous step suggest that BusyBox is characterized by a high degree of visibility. The browsing of the source code is facilitated by the availability of a subversion system (SVN). Source files are packaged in 28 main directories and information about number of revisions, authors of the revisions, age of the latest revisions, and log entries is provided for each directory. This facilitates the applicability of all the *white-box testing* techniques and a clear identification of the units that compose the entire system. The high level of modularity and the low level of interoperability among the features of BusyBox seem to suggest that developers should focus on unit testing activities.

Non-functional issues are not of primary importance for this kind of tool, due to its nature: BusyBox is a tool provided without a graphical interface thus approaches such as *capture and replay* are infeasible; the use of monitors that probe memory usage and execution/response time are not meaningful since the tool only provides calls to simple functions. Finally, since BusyBox provides replacements for most of the utilities usually found in GNU, developers should pay attention to testing security aspects. It is realistic to imagine a scenario in which a developer inserts malicious code into a BusyBox function to remotely control the operating system of an end-user that has installed BusyBox. This requires the execution of *acceptance tests* that check the main functionalities of the tool in order to admit only trusted behaviours. For example, a test case should verify that the Unix command `su` (`superuser`) must not record the typed password.

As for issues I2, I3, I5, the BusyBox project is not supplied with project plans, documents that describe the system requirements analysis, risk analysis, technical documents that describe the use of standard protocols or patterns, architectural models, etc. The only standard to which developers pay attention, without completely adhering to it, is the “Shell and Utilities” portion of the Open Group Base Standards. This strongly limits the applicability of all testing solutions that are based on project specifications such as *model-based* and *conformance testing* techniques. However, the web portal of BusyBox provides a section that describes all the features and functionalities offered by BusyBox in a structured way. Each feature description reports the input and output parameters, the behaviour the feature should have, and how to use the feature. This fosters the applicability of black-box techniques such as *Category Partition* and *Catalog-based testing* techniques in addition to white-box testing techniques. Moreover, developers and final users should share testing knowledge with each other. Unit tests, system tests, and regression tests results should be provided to the global community to favour and simplify BusyBox's integration testing activity.

As for I4, BusyBox is a vital and consolidated project (latest revisions are usually few days old and the analyzed release is V1.14.0) supported by a small/medium-size community of developers (currently, 38 accounts exist on busybox.net). It is characterized by a collaborative development process and rapid system growth: at the time of writing, 24.110 revisions have been performed by the community and forums and mailing lists are still alive and fruitful. Also, the bug tracking system seems to indicate an active community of developers (at the time of writing, several bugs have been recently fixed, some bugs are unassigned and are waiting to be fixed, and the mean time required to fix a bug is quite short: 3/4 days). All of these considerations seem to suggest the need for a strong regression testing activity during the whole development process of the BusyBox tool, to avoid bugs introduced by a lavish creativity.

Summarizing, the best testing process for BusyBox should take into account: (1) unit testing activities; (2) integration testing activities; (3) regression testing activities; (4) security testing activities; (5) acceptance/system testing activities; (6) the identification and use of test management tools; (7) the documentation and sharing of test results.

Step 3: ATP analysis

Currently, BusyBox is released along with a test suite that can be executed by final users and developers to identify problems and bugs when BusyBox is installed on machines different from the tested ones. A quick look at the test suite suggests that developers have designed test cases to only stress each feature of BusyBox separately. Unfortunately, reports and documents that discuss the test cases and the execution results are not provided. The testing plan provided for BusyBox does not have the ability of automatically logging and collecting the test results, and users must manually signal potential bugs to the community.

Step 4: Maturity level evaluation

Comparing the BTP derived for BusyBox during Step 2 and the one currently available, it is clear that the testing activity of BusyBox is actually poor (this is also confirmed by the huge number of bugs posted by the community). The intersection between BTP and ATP does not exceed 25% (only 1 activity out of 7 is currently supported by the ATP), thus BusyBox maturity level is *ML=1*. This suggests the need for applying all the testing guidelines previously identified by our methodology to increase the quality and the trustworthiness perception of the tool.

Step 5: Testing process improvement

To improve the testing process of BusyBox, we selected a test management tool (*TestLink*) to create and manage test cases and test plans, execute test cases, track test results dynamically, and generate reports.

We planned integration, acceptance/system and regression testing activities, and then we generated a set of test cases for each activity. Then, we executed the test suites on a machine hosted in our lab with the following environment: 2 CPU Intel Xeon 3.73GHz (cache size 2048KB); 8GB RAM; 250GB Hard Disk; Gentoo-r6 Linux distribution; Kernel 2.6.18; C compiler: gcc 4.1.2 with glibc 2.3. reports the data collected during the execution of the test suites. Column <*BusyBox*> reports the version of BusyBox under test; Columns <*#of TCs*>, <*#of Passed TCs*>, <*#of Failed TCs*> report the total number of test cases for each test suite, the number of test cases that succeeded (i.e., that ended with no failures), and the number of test cases that failed (i.e., that resulted with the software having a failure), respectively.

System/Acceptance testing has been performed by executing three different sets of test cases against three different versions of BusyBox (Table 10 (a)). A simple example of test case is *test (pwd) = (busybox pwd)*: this test case simply verifies that the working directory returned by the operating system is equal to the one returned by BusyBox. The execution of the test cases allowed to discover a lot of incompatibilities between BusyBox implementations of some UNIX utilities in combination with the hardware and software environment used

for the experiments.

Regression testing has been performed by re-executing the 358 successful test cases, designed for BusyBox V1.12.1, against the new stable releases V1.12.4, V1.13.0 and V1.13.2 (Table 10 (b)). When executing the test suite against BusyBox V1.12.4, none of the test cases failed. However, when executing the test suite against the BusyBox V1.13.0, three test cases (that refer to the `taskset` command) were not executed because `taskset` has been removed from this version of BusyBox, and just one test case failed. An analysis of the test result allowed us to identify a new error introduced by a code change (related to the `cpio` command) in V1.13.0. The error even persisted in BusyBox V1.13.2.

Integration testing has been performed by executing 48 test cases designed to check the interoperability between the BusyBox implementations of the most common UNIX utilities (e.g., `cp` command in combination with `touch` and `cmp` commands). In this experiment, two test cases failed. The first one failed due to an unsupported option (`-t`) for the `od` command when piped with the `echo` command; the second one failed due to an unsupported option (`--date=`) for the `touch` command when piped with the `mv` command.

This experiment demonstrates the simplicity of our Maturity Model, and also the real benefits introduced by a well planned testing process. The activity of applying the OSS-TMM to detect the maturity level of BusyBox (Step 1 to Step 4) required a limited effort (one skilled person fully worked one day for this task). The activity of restructuring the testing process of BusyBox following the suggestions provided by OSS-TMM (Step 5) required a strong effort (one skilled person fully worked one week for this task). However, the restructured process provided the ability to detect three new errors in BusyBox V1.13.2 with an actual improvement of the BusyBox quality.

Table 10 – Test cases results for BusyBox

Acceptance/System Testing

BusyBox	#of TCs	#of Passed TCs	#of Failed TCs
V1.10.1	312	291	21
V1.12.1	387	358	29
V1.13.2	390	359	31

(a)

Regression Testing

BusyBox	#of TCs	#of Passed TCs	#of Failed TCs
V1.12.4	358	358	0
V1.13.0	358	354	1

V1.13.2	358	354	1

(b)

Integration Testing

BusyBox	#of TCs	#of Passed TCs	#of Failed TCs
V1.13.2	48	46	2

(c)

3.5.2. Apache HTTP Evaluation

Here, we apply OSS-TMM to the Apache HTTP project. We applied the OSS-TMM to Apache HTTP from the final user point-of-view with the aim of demonstrating how a non-skilled user can derive the maturity level of the product he/she is evaluating.

Apache HTTP [`http://httpd.apache.org/`] is an open source HTTP server for modern operating systems such as UNIX and Windows, which provides HTTP services in sync with the current HTTP standards.

As in the previous case study, we sequentially applied all the steps of our methodology, with the only exception that we focused on the steps followed by a final user interested in evaluating the Apache HTTP product. Hence, we first analyzed Apache HTTP through our checklist, we identified the *BTP* in relation to the actual characteristics of Apache HTTP, and we estimated the maturity level of the product by comparing the *BTP* and the Apache *ATP*.

Table 11 summarizes the Apache HTTP characteristics derived from the analysis. We only report the issues that actually characterize Apache and are useful to derive the BTP.

Table 11 – Step 1 outcome for Apache HTTP

Issue	Apache HTTP characteristic
I1.1	Apache is managed via an Historical Archive
I1.2	the whole project is well structured
I1.3	information about releases are visible
I1.4	information about code revisions are visible
I1.5	SSI and AAA modules are security critical
I1.6	all the test scripts are open source to the community
I1.8	*access_log*, *error_logfiles* are collectable
I2.7	coding standards are specified in a *style guide*
I2.8	performance constraints: *resource usage*, *latency*, *throughput*,

	scalability
I2.11	external modules: *mod_python*, *mod_ftp*, *mod_mbox*
I3.5	The *Apache-Test Framework* is recommended
I3.6	the *Bugzilla* bug tracking system is integrated
I4.1	the number of revisions is huge
I4.2	more than 100 developers/contributors
I4.3	the analyzed release is: V2.2.11
I5.8	installation requirements are documented (but not up-to-date)
I5.9	test documentation is unavailable

Starting from the Apache characteristics highlighted in the previous step, the best testing process for Apache should take into account: (1) unit testing activities; (2) integration testing activities; (3) regression testing activities; (4) security testing activities; (5) performance testing (load testing, stress testing, endurance testing, etc.); (6) versioning compatibility checks; (7) acceptance/system testing activities; (8) source code inspection through checklists for C/C++; (9) use of test management tools; (10) installation testing activities; (11) documentation and sharing of test results.

Currently, the source code of Apache HTTP is released with a small test suite that tests the critical features of the project. Moreover, the project supports the *SPECWeb99 benchmark*, *Flood subproject*, and the *Apache-test framework*. SPECWeb99 and Flood can be used to gather important performance and security metrics for websites that use Apache HTTP. The Apache-test framework supports the definition of test suites for products running on the Apache HTTP, and can be used to run existing tests, setup a testing environment for a new project, and develop new tests. However, a complete test suite for integration and regression testing is not provided and also source code inspection, and versioning compatibility checks are not yet performed. The requirements 1, 4, 5, 7, 9, 11 of BTP are addressed by the ATP of Apache HTTP, thus implying an Apache Maturity Level of *ML=3*.

This case study demonstrates how simple it is to apply (with a minimal effort) the OSS-TMM to complex projects as well, as in the case of Apache HTTP. The checklist strongly simplifies and supports the analysis by suggesting step-by-step activities that non-skilled people can follow to determine the maturity level of the product under evaluation.

3.5.3. Other Evaluations

Finally, we applied the OSS-TMM to four additional OSS projects to evaluate their maturity level, and we look for correlation patterns between the obtained score with dependent measures to comprehend whether a high maturity of the

testing process directly means a high quality of the product, and a low maturity directly means a low quality.

We selected the four OSS projects by evaluating their size, their organizational type (i.e., sponsored, foundation, spontaneous) and their diffusion to identify a heterogeneous set of OSS projects. We selected: the Debian distribution (DebianOS) as a well known, sponsored, and complex OSS product; the Data Display Debugger (DDD) as an unfamiliar, sponsored, and a project with reduced complexity; the OSS database PostgreSQL as a specialized, sponsored, and complex product; the web content management system (Xoops) as a specialized, founded project of reduced complexity.

As for the dependent measure, we decided to select the bug rate (*BR*) of the product (i.e., the number of bugs divided by the product size in thousands of lines of code KLOC), which is a reliable indicator of the overall product quality. Table 12 resumes the obtained results. Bugs data have been collected by analyzing the bug tracker system of each product with focus on open bugs. We selected the latest stable release of each product to avoid strange bug distributions related to newly released and unstable products. We used SLOCCount (developed by D. Wheeler) for counting the physical source lines of code of each project [www.dwheeler.com/sloccount/].

Table 12 – Maturity Level (ML) and Bug Rate (BR) for six OSS products

OSS Project	ML	SLOC	BUG	BR
Apache HTTP v2.2.0	3	135916	19	0.14
BusyBox v1.13.2	1	177013	9	0.05
DDD v3.3	1	119194	14	0.12
DebianOS v3.0	1	104679026	10968	0.11
PostgreSQL v8.3	4	909148	37	0.04
Xoops Core v2.3	2	74551	35	0.47

The collected data do not completely confirm our initial hypothesis (i.e., a better testing process always means a higher quality of the product and vice versa). In effect, projects with a very low ML (such as Debian, DDD, and BusyBox) have a very low bug rate; projects with a medium ML (such as Xoops) have a high bug rate; and projects with a high ML (such as Apache HTTP and PostgreSQL) have a low bug rate too. Hence, the collected data partially confirm our hypothesis: a well planned testing activity favours the overall quality of the product. Of course, the sample used in this experiment is limited in the number of products, but it is an interesting starting point for additional experiences. Moreover, the bug rate is computed by analyzing the bug tracker system of each product, thus introducing variability due to the quality of the published

data. For instance, we detected a discrepancy in BusyBox between our test cases results (see Table 9) and the number of bugs reported by the bug tracker system (see Table 12). Further experiences should take into account additional measures such as the defect rate of the products. Currently, we are correlating the collected *MLs* with the code coverage provided by the available test suites of each project under analysis in order to validate the results obtained in Table 12.

OSS-TMM has also been internally applied at Siemens AG to evaluate the testing maturity level of TPTP. The evaluation has been carried out twice. Firstly, we applied the OSS-TMM checklist and we computed its ATP surfing the information stored into the TPTP repository. The second evaluation has been carried out interviewing TPTP project leaders and developers. The first evaluation (repository based) resulted in a maturity level ML=2 (intersection between ATP and BTP is equal to 34,78%) and the second evaluation (TPTP project lead/board members) resulted in the same maturity level (with an intersection equals to 47,83%). In this way, we validated the results of the first evaluation.

3.6. Related work

We compare OSS-TMM with what has been done in some related research areas that address software quality management.

Software Process Improvement

Research in software process improvement focuses on certification models that deal with the quality of the software production process. The most important models are CMM and SPICE [Herbsleb, et al. 1997] [ISO2 2004]. The Capability Maturity Model (CMM), and its extension CMMI, is a methodology that assists companies in understanding the capability maturity of their software processes. The maturity model involves several aspects related to five maturity levels (chaotic, repeatable, defined, managed, and optimizing), a cluster of Key Process Areas (KPA) (i.e., related activities that, when performed collectively, achieve a set of important goals), a set of goals (i.e., scope, boundaries, and intent of each key process area), common features (i.e., practices that implement a KPA), and finally key practices (i.e., the elements that effectively contribute to the implementation of the KPAs).

The Software Process Improvement and Capability dEtermination [SPICE / ISO15504] is a framework for the assessment of processes. The SPICE reference model focuses on a wider vision than CMM by taking into account five process and capability dimensions (customer-supplier, engineering, supporting, management, and organization). In compliance with CMM, they define a scale of capability levels, a cluster of process attributes (to measure capability of processes), a set of generic practices (i.e., indicators to aid assessment performance), and a process assessment guide.

Our approach is built upon the general ideas proposed by CMM and SPICE.

However, OSS-TMM uses a simpler and less rigid maturity model than CMM and SPICE, because of its purpose and its focus on the testing dimension. Moreover, the OSS-TMM process assessment also suggests how to improve the available testing process of an OSS product by recommending the most suitable testing techniques.

Software Product Quality

The most important standard to ensure the quality of the product is ISO9126 [ISO3 2001]. ISO9126 standard takes into account several aspects of the internal, external, and in-use quality of a software product and it defines a quality model that includes a set of characteristics and sub-characteristics related to functionality, reliability, usability, efficiency, maintainability, and portability. In ISO9126 a wide set of complex measures are defined to assess product quality, while ISO14598 [ISO1 2001] provides an explanation of how to apply the ISO9126 model.

Our approach focuses on the quality of the testing process instead of the whole product quality and it simplifies the evaluation of the process maturity by providing a checklist instead of a complex list of measures. The steps that compose the OSS-TMM process assessment are compliant with the guidance and requirements for software evaluation highlighted in ISO14598.

Testing Maturity Models

Research in testing maturity models complements CMM with the focus on testing aspects. The first work on this research area is provided by Burnstein et al. in [Burnstein, Suwanassart and Carlson 1996]. They defined a Testing Maturity Model (TMM) that helps evaluate the testing process of software products. TMM identifies five rigid maturity levels, a set of maturity goals and sub-goals (equivalent to KPAs of CMM), and a set of activities, tasks and responsibilities (ATR) for each maturity level.

Other CMM-based testing models have been proposed. For example, the Test Improvement Model (TIM) [Ericson, Subotic and Ursing 1997] and the Test Process Improvement Model (TPI) [Koomen and Pol 1999] suggest ways in which testers can improve their work. TIM and TPI identify key areas for the testing process starting from the organization and planning of testing activities to test cases generation, execution, and documentation review.

While the previous approaches have been designed with CSS characteristics in mind, OSS-TMM exploits the inherent characteristics and issues typical of OSS products. Hence, OSS-TMM defines four maturity levels that are not structured as sets of predetermined maturity characteristics and goals, but they depend on the actual characteristics of the product under evaluation. Moreover, OSS-TMM supports both testers in improving the testing process and also companies and final users in assessing the quality and the trustworthiness perception of the OSS product.

OSS Quality Assessment

Research in OSS quality assessment extends CMM and CMM-compliant models to identify, from the set of CMM goals, only the subset that is relevant for OSS products. The first CMM extension for OSS is the Open Source Maturity Model (OSMM) [Duijnhouwer and Widdows 2003]. OSMM defines a methodology and a set of OSS *ad hoc* indicators to assess the global maturity of an OSS product, helping final users to choose between equivalent OSS products. Since the definition of OSMM, several other models have been developed see as example [Taibi, Lavazza and Morasca 2007].

OSS-TMM does not provide a global assessment of the product quality but uses the testing process maturity level as an indicator of the process quality. This simplifies the applicability of the approach and the identification of weaknesses into testing processes.

3.7. Final remarks

We have outlined the levels of a new Maturity Model (OSS-TMM) for the testing process of OSS projects and we have described the goals it helps reach and the issues involved. Applications to BusyBox, Apache HTTP and TPTP show how the issues come into play on real-life projects. Having a Maturity Model for the testing activities of OSS processes may even be more important than in CSS. OSS processes are usually much less structured than CSS processes and may be considered closer to Agile development and XP in many respects.

The approach needs to be applied to more OSS projects, to gather enough information about its actual effectiveness in several domains. We believe that continuous gathering and analysis of experience will help pinpoint specific issues of OSS testing and better address the building of a more refined and useful OSS Testing Maturity Model. Moreover, the correlation between the maturity levels of the project under analysis with the code coverage of their test suites will provide another important indication about the validity of our approach. We are also investigating whether there is a correlation between classes of OSS products (such as operating systems, middleware systems, CMSs, etc.) and common best practices to test these classes of products. This will help the identification of common automatic testing supports that could simplify the testing process of these classes of products.

4. T-DOC FRAMEWORK

In this section, we introduce the T-DOC framework. The presentation is organized as follows: first, a report of the analysis confirming the low availability of testing documentation is presented; the motivations that are at the basis for adopting built-in testing in the context of OSS products is discussed next; then, the T-DOC framework and how it comes into play when applied to two case studies.

4.1. The Lack of OSS Documentation

The perception we normally have surfing the web portal of OSS products, observing OSS forums/blogs/discussions, and using OSS products in our every-day work is that most of the available OSS projects are released without user manuals and technical documents.

To have an empirical evidence of this perception, we conducted a two-fold analysis: first, we interviewed 151 OSS users (end users, developers, managers, OSS experts) and then, we analyzed the web portal of 32 well-known OSS projects. An extensive report of these experiences can be found in [QualiPSo1 2009] [QualiPSo2 2009]. The first analysis aimed to identify the importance the factor "availability of technical documentation / user manual" have for OSS users. We discovered that in a scale from 1 (negligible importance) to 8 (fundamental importance), the factor "availability of technical documentation / user manual" took a very high score equal to 6.5. The second analysis aimed to check the actual availability of technical documentations and user manuals related to the 32 analyzed projects. We discovered that: 69% of the projects have up-to-date user manuals while the remaining 31% have not updated or available user manuals; 49% of the projects have an up-to-date technical documentation, while the remaining 51% have not an updated or available technical documentation.

This deficiency is exacerbated if we look at testing documentation: in our analysis, only 1 product (out of 32) provides a complete documentation about its internal testing activities. Only JBoss [www.jboss.org] exposes a detailed and up-to-date documentation about testing plans, testing methodologies, test cases description, and test suite results. We believe that this is primarily due to three main reasons: first, the use of classical testing methodologies that are based on external testing (i.e., test cases are independent components that are separated from the applicative code) drastically augment the fragmentation of data, thus further complicating the process of documenting testing activities; second, the lack of well-agreed best practices on how to test OSS products increases the effort required for testing applications, thus stealing effort in documenting testing activities. In this document, a methodology to assess and improve the testing process of OSS products is presented in Section 2.6; finally, the lack of tools, which support and automate the documentation of testing activities, leaves too much effort to the side of developers.

This analysis confirms our intuition and demonstrates the need for a framework, based on built-in testing, that supports the automatic generation of testing documentation. The next section discusses why a built-in testing methodology is preferred to external testing solutions.

4.2. Built-in test in OSS

Built-in test (BIT) approaches for software systems originated in the context of component-based systems both to simplify the integration of third-party black-box components and also to enhance software maintainability [King, Wang and Wickburg 1999]. A BIT component (or BIT class) is a traditional component that puts together application code with testing code [Beydeda 2005]. A BIT component can operate in a normal mode (i.e., testing capabilities are transparent to the user) or in maintenance mode (i.e., the user can test the component in his environment by exploiting the built-in testing capabilities) by interacting with the normal or the testing interface, respectively. Listing 1 shows a code excerpt for a typical component with built-in testing abilities, where test cases are declared and implemented directly into the applicative class.

```
Class class_name {
      //normal interface
      Data declaration;
      Constructor declaration;
      Destructor declaration;
      Methods declaration;
      //testing interface
      Tests declaration;
      //normal implementation
      Constructor;
      Destructor;
      Methods;
      //testing implementation
      Test cases;
}
```

Listing 1 – Code excerpt of a BIT component.

In the context of OSS, the heterogeneity of the developers/contributors increases the fragmentation of the source code and makes unfeasible the adoption of common external testing methodologies, programming rules, and testing tools that could favour the whole comprehension of fragmented testing activities. Keep in mind simple programming rules (as shown in Listing 1) can

favour the standardization of a common programming style that can improve the testing activity, decrease the effort spent in testing, and simplify the generation of testing documentation. Whenever a developer/contributor of an OSS product introduces or modifies a functionality of a component, she designs and codes unit tests, integration tests and optionally non-functional tests into the component to provide BIT abilities. Modified components are then uploaded into the repository that stores the project and are integrated to generate the OSS product with comprehensive BIT abilities (as shown in Figure 6).

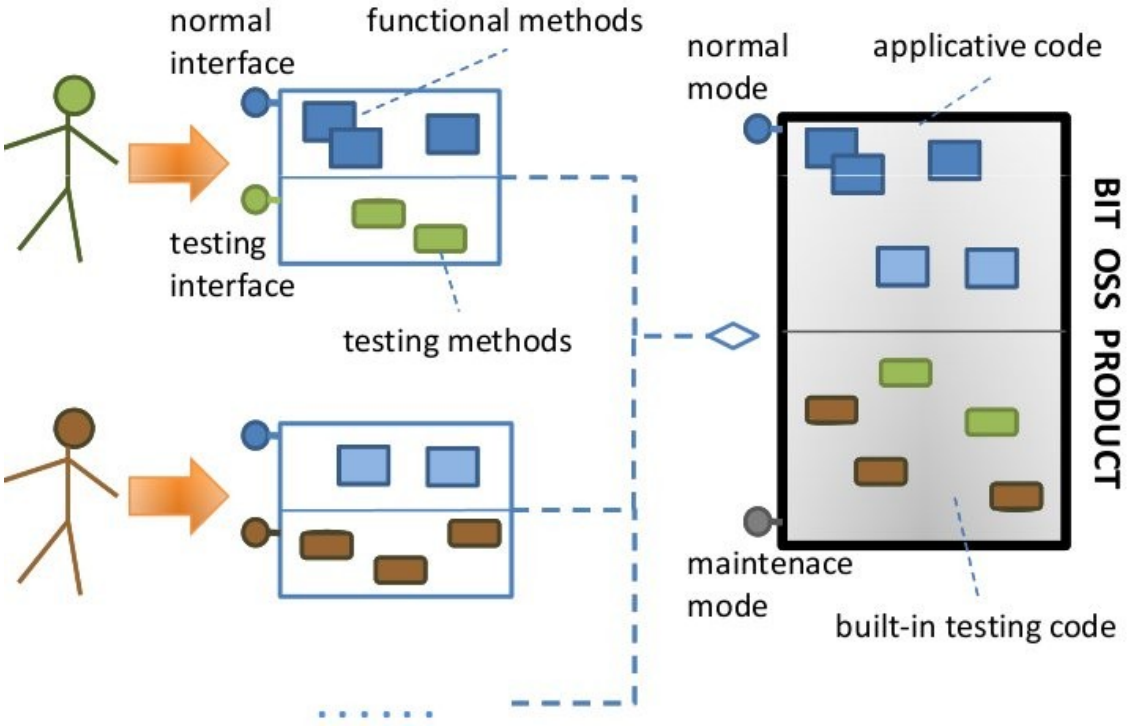

Figure 6 – Aggregating components into an OSS product with BIT abilities.

Putting together application code and testing code into single classes improves the visibility and inheritance of test cases, it favours the standardization of testing interfaces, and it augments aggregation of data, thus simplifying the discovery of testing data and the correlation with coding elements. Moreover, the documentation of test activities and the report of test case results is made easier, thus simplifying regression testing activities. BIT favours run-time testing: the system can be executed at run-time in maintenance modality [Suliman, et al. 2006], thus simplifying the detection of bugs that are undetectable in the controlled testing environment. Moreover, the test suite can be executed over different HW/SW platform configurations, thus simplifying system, configuration and performance testing. Hence, the "eye bird" ability, which is typical of OSS products (i.e., the capacity to evaluate a product by the large glance of the OSS community), can be fully exploited and can be complemented by testing activities.

However, BIT also introduces risks and limitations that need to be faced when

designing the T-DOC framework: run-time testing can move the system in an inconsistent state that may compromise the stability of the system. To mitigate this risk, the test suite must be executed in background only once, during the OSS product installation (or during critical updates). Moreover, BIT is an intrusive mechanism that can lead to security and privacy-related problems. To mitigate this risk, final users must be advised about the BIT abilities of the OSS product, so that they can block the BIT abilities, and user-related data must not be collected by the framework. Finally, if built-in tests are executed without a control, system performance can degrade. The execution of the built-in test suite in background, during the OSS product installation, alleviates this problem.

To the best of our knowledge, we believe that the use of BIT abilities, instead of external testing mechanisms, is the only way to support and simplify the generation and the gathering of testing documentation in the domain of OSS.

4.3. The T-DOC framework

We present the T-DOC framework and we detail its threefold support by separately discussing:

- the automatic generation of test cases documentation,
- the automatic generation of suggestions about integration and regression testing activities, and finally,
- the generation of reports about the results of the test suite execution.

4.3.1. Test cases documentation

This first layer of support aims at simplifying and supporting the automatic generation of the documentation about test cases and test suites. The generated documentation should increase the readability of the technical aspects of each test case, and should favour an overall comprehension of the testing activity. To allow the automatic execution of this process, built-in test cases must be surrounded by doc comments (i.e. short sentences that describe the test case, its purpose and its behaviour) and keywords in a similar way comments and *block taglets* surround methods and functionalities in Java source code. Testing doc comments (T-DOC comments) and *block taglets* are then parsed and elaborated by the T-DOC engine to generate the test case documentation in a similar way the Javadoc tool operates.

Javadoc is a tool from Sun Microsystems for generating API documentation out of declarations and documentation comments in Java source code. Javadoc produces HTML documentation describing the packages, classes, interfaces, methods, etc. of a software system. The output format of the Javadoc can be customized by means of *doclets*. Javadoc parses special tags embedded within a Java doc comment. These doc tags are used to automatically generate a

complete, well formatted API from the source code. All tags start with a (@), e.g., @author. The tags are used to add specific information like a method's parameters (@param), return type (@return), and exceptions (@exception).

To minimize the effort of developers and contributors in writing testing documentation, to favour standardization, and to avoid subjective interpretations of data, we clearly define a set of new conventions and a set of new tags that developers and contributors should follow whenever they add a T-DOC comment. An example of a real T-DOC comment can be found in Figure 7.

```
public void applyAction() {
    currentPlayer.setMoney(currentPlayer.getMoney()+amount);
}
/**
 *   Tests the behavior of the applyAction() functionality. Checks whether the account of
 *   the current player's CCard is properly updated when a gain of money is performed.
 *
 *   @succeedIf      getMoney() returns a value = 1550 $
 *   @failIf         getMoney() returns a value != 1550 $
 *   @typology       unit testing
 *   @author         Davide Tosi
 *   @version        1.0.2  06/02/09
 *   @see    edu.ncsu.realestate.MoneyCard.MoneyCard()
 */
public void testGainMoneyCardAction() {
    Card gainMoney = new MoneyCard("50$", 50, Card.CARD_TYPE_CHANCE);
    GameMaster.instance().getGameBoard().addCard(gainMoney);
    card.applyAction();
    TestCase.assertEquals(origMoney+50, GameMaster.instance().getCurrentPlayer().getMoney());
}
```

Figure 7 – A built-in test case with T-DOC comments for the RealEstate application.

The defined conventions are:

1) the first line contains the begin-comment delimiter (/**)

2) write the first sentence as a short summary of the test, as T-DOC engine automatically places it in the summary table of the test

3) insert a blank comment line between the description and the list of tags

4) the first line that begins with an "@" character ends the description

5) there is only one description block per T-DOC comment

6) the last line contains the end-comment delimiter (*/)

The new defined tags are:

@param (name of the parameter, followed by its description)

@return (omit @return for tests that return void; required otherwise)

@succeedIf (summarize the conditions under which the test case succeeds)

@failIf (summarize the conditions under which the test case fails)

@qualityAttribute (specify the quality category addressed: performance, security,...)

@scope (specify the test case purpose: unit, integration, structural)

```
@author (author name/surname)
@version (version number + checkout date)
@see package.Class#method(Type,...)(ref to the functionality under
test)
```

Figure 8 shows a subset of the functionalities provided by the T-DOC engine. The T-DOC engine takes in input the set of classes that are added/modified by the developer. Each class is analyzed separately to discover and isolate the built-in test cases and their T-DOC comments. The `Test Suite Builder` component aggregates all the built-in test cases into a single test suite, and the `T-DOC TCs` component parses all the t-doc comments to generate the complete documentation of the test suite. Finally, the engine publishes the documentation to the central repository (`Test Tracker`) of the project to avoid fragmentation and versioning problems of the documentation. Versioning problems are also avoided by means of the introduction of the new tag `@version`.

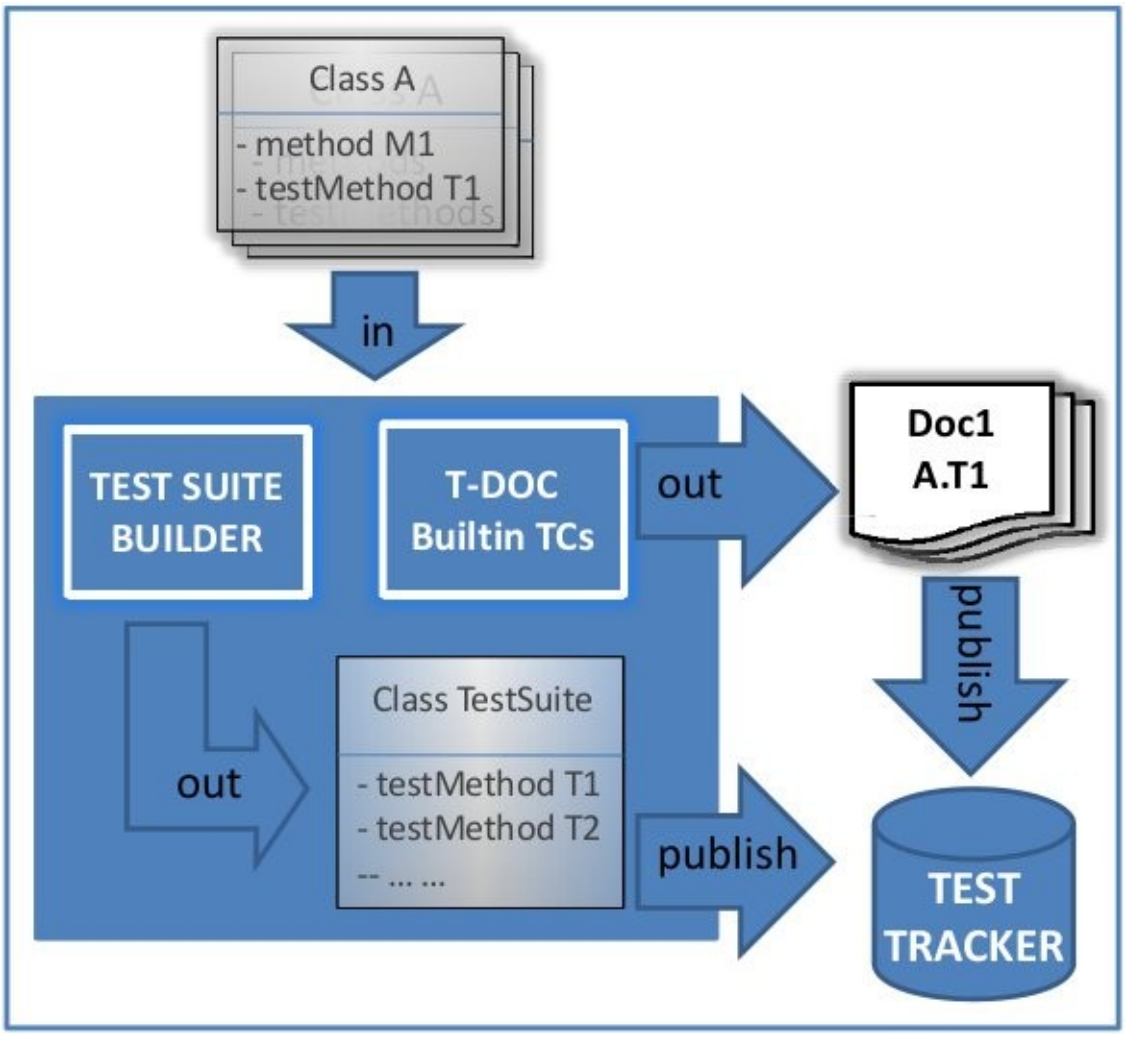

Figure 8 – Architecture of the first T-DOC layer.

To favour the comprehension of this layer, we exemplify the writing of a T-DOC comment for a built-in test case we derived for the RealEstate OSS Java

application [NCSU 2009], which will be used as proof-of-concept of our work. Figure 7 shows the source code of the built-in test case surrounded by a T-DOC comment and T-DOC tags. Purpose of this figure is not to present the internal code of the test, but to highlight the structure of a T-DOC comment.

The documentation automatically generated by the T-DOC engine for the aforementioned test case looks like as follows:

```
ID001:: UNIT Test: testGainMoneyCardAction

V1.0.2 05-26-09

Tests the behavior of the applyAction()functionality. Check whether
the account of the current player's CCard is properly updated when a
gain of money is performed.

Succeeds if: getMoney() returns a value = 1550$

Fails if: getMoney() returns a value != 1550$

See: edu.ncsu.realestate.MoneyCard.applyAction()
```

The T-DOC engine generates a documentation that is compliant with the visual representation of Javadoc comments, with small differences (such as the use of a label for each test `ID00X`), in order to maximize both the compatibility between the tools and also the readability of the documentation.

In Section 4.3.4, we present a thorough case study of the application of the first layer of T-DOC. We have utilized it during the development of MACXIM – a tool to collect metrics from Java code.

4.3.2. Regression and Integration testing documentation

This second layer of support aims at suggesting and documenting the integration and regression test cases that OSS contributors should develop during the update/maintenance of their OSS products. The generated documentation should simplify the contributors' task of writing these test cases. To this end, the critical dependencies among methods and components must be detected by the T-DOC engine and visually reported to the developer. The T-DOC engine exploits and extends the idea of `change points` and `call graphs` [Mao, Lu and Zhang 2007] [Orso, et al. 2001] to automatically detect the source code location in which a code change has been performed, and to automatically create the graph of calls related to the method in which the change has been detected. These graphs are used by the T-DOC engine as starting point to create the suggestions for integration and regression testing activities.

Figure 9 shows a subset of the functionalities provided by the T-DOC engine. This layer of the T-DOC engine is composed of three main modules: the `T-DOC Integration` module, the `T-DOC Regression` module and the `Call Graph` tool.

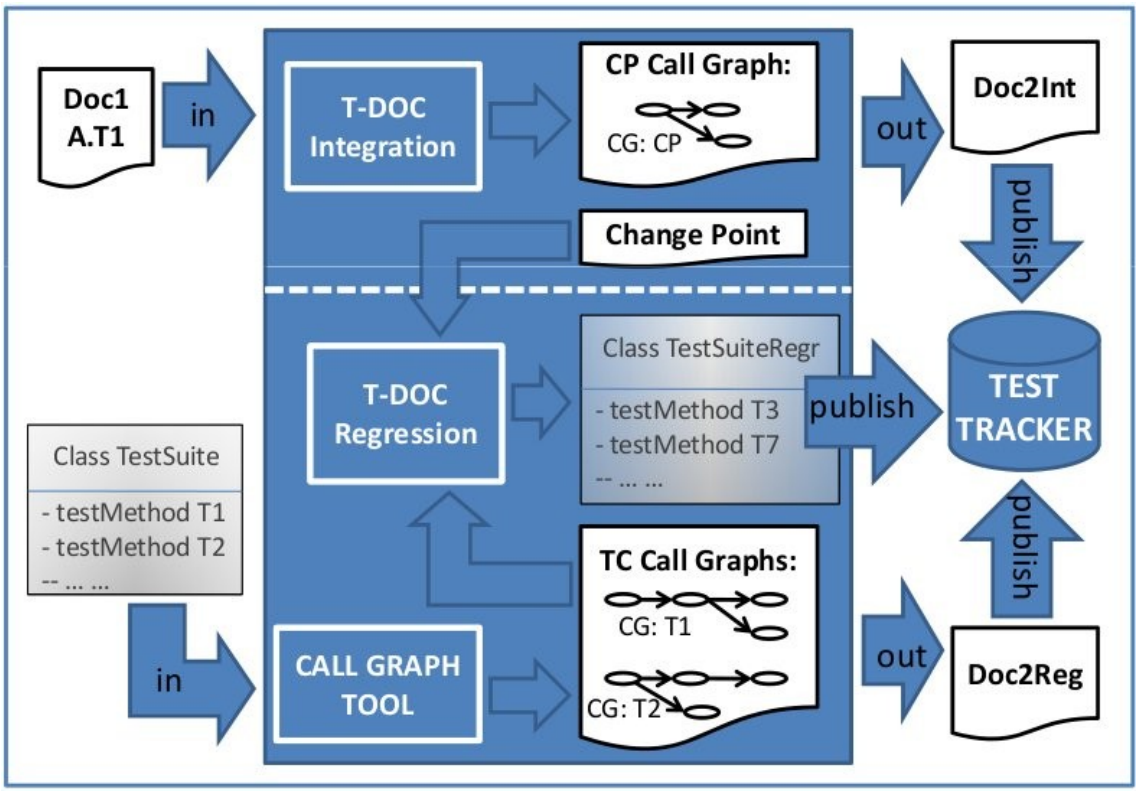

Figure 9 – Architecture of the second T-DOC layer.

`T-DOC Integration` module is responsible for suggesting integration testing scenarios that should be implemented by the OSS contributors whenever a new method is added or whenever an existing method is modified (i.e., the `@version` tag of the associated test case is updated). Integration testing checks dependencies among objects of different classes. Class A and B are related if objects of class A make method calls on objects of class B, or if objects of A contain references to objects of B. The `T-DOC Integration` takes in input the documentation generated by the `T-DOC TCs` module (`Doc1 A.T1`), and it generates the call graph for the change point (`CP`) that is related to the documented test case. To avoid the explosion of the graph size, we limited the computation to the third level of method's dependencies. Referring to our RealEstate example of Figure 7, the OSS contributor is working on the `MoneyCard` class. He is modifying the `applyAction()` method, and he is writing the built-in test case `testGainMoneyCardAction()`. First of all, the `T-DOC Integration` module computes the call graph for the change point `applyAction()`, then it produces the integration testing scenario for this change. Figure 10 shows the result of this computation (`Doc2Int`). The root of the graph is the CP `applyAction()`, while leafs are the methods that directly or indirectly interact with the `applyAction()` method.

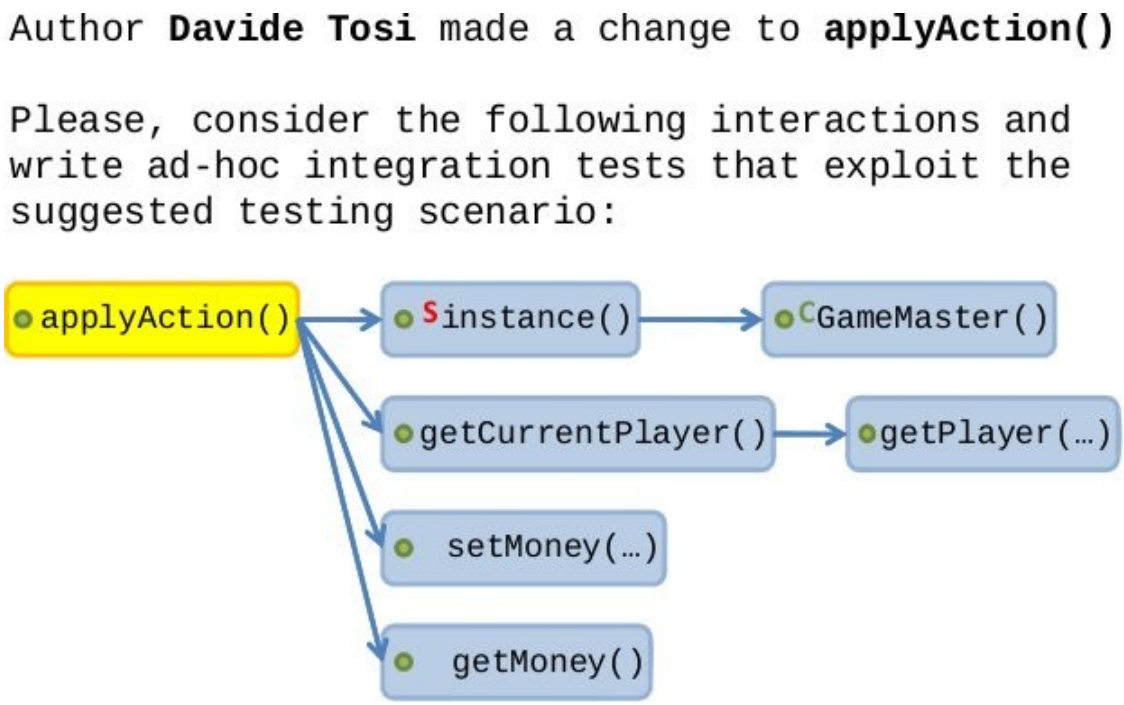

Figure 10 – Generated integration testing scenario for the `testGainMoneyCard Action()`.

The `T-DOC Regression` module is responsible for automatically detecting the subset of relevant test cases for regression activities whenever a change into the code is performed. Without this support, OSS contributors are forced to manually rerun all the test cases in the test suite for regression purposes. This task is very expensive for contributors that are not interested in testing. For instance, rerunning the complete test suite for the OSS WEKA application [`www.cs.waikato.ac.nz/~ml/weka/`] require 45 minutes in a fully dedicated machine. Moreover, other problems are: who runs the test suite? Where does one store the test cases that should be re-executed? When must the test cases be rerun? Where are reported the results of the test suite execution? All these problems are addressed by the `T-DOC Regression` module. This module takes as input the change point and also the complete set of call graphs computed for each test case by the `Call Graph Tool` module. Then, the `T-DOC Regression` module scans all the call graphs to detect the subset of graphs that are affected by the change point (i.e., the change point is present into the graph). The subset of relevant call graphs indicates the meaningful test cases that should be re-executed with respect to the change that has been performed. The algorithm that the `T-DOC Regression` module computes for detecting the subset of meaningful test cases is outlined next:

```
Input: test cases, CP
Output: documentation of the subset of meaningful regression test
cases
1. derive the call graph for each test case stopping at the third
level of dependencies;
```

```
2. select a graph as starting entry;
3. scan the graph in order to detect whether the change point is
present;
4. if the change point is present: select the test case for
regression;
   else: jump to step 2.
5. when all the graphs have been evaluated, generate the regression
documentation as the list of test cases wrt the CP
```

For the RealEstate application the `T-DOC Regression` module takes in input, from the `Call Graph Tool`, 30 graphs and generated the following documentation (`Doc2Reg`):

```
This is the subset of regression test cases for the applyAction()
change point:
01) testGainMoneyCardAction()
02) testMovePlayerCardAction()
03) testLoseMoneyCardAction()
04) testJailCardAction()
05) testJailCardUI()
06) testLoseMoneyCardUI()
07) testMovePlayerCardUI()
```

For space reason, we do not show the complete set of graphs computed by the T-DOC engine. Moreover, in this paper, we do not provide the empirical evidence that the coverage obtained by the subset of the selected test cases is actually the best one. We are conducting additional experiments in this direction.

All the data provided by this second layer (Doc2Int, Doc2Reg and the regression test suite) are published into the central Test Tracker system.

4.3.3. Test cases execution report

This third layer of support aims at homogenizing and collecting both all the outputs coming from the T-DOC framework and the results obtained by the execution of the test cases. In this section, we only introduce the design of this layer since an implementation is not yet available. This layer is composed of two main entities: the `Test Tracker` system and the part of the T-DOC engine that is responsible for collecting and manipulating the test case results.

The `Test Tracker` system is responsible of managing: (1) the class containing all the built-in test cases that are incrementally added (or modified) to the test suite (`Class TestSuite`); (2) the class of integration test cases (if

available); (3) the class containing the regression test cases derived by the `T-DOC Regression` module. The `Test Tracker` system stores the documentation of each test case (`Doc1 A.T1`, `Doc1 A.T2`, `Doc1 A.Tn`) and aggregates this documentation in a single document that describes the complete behaviour of the test suite. Moreover, the `Test Tracker` system stores the documentation related to integration and regression test cases (`Doc2Int` and `Doc2Reg`), and it aggregates this documentation in a single file. Finally, the `Test Tracker` system provides search abilities among all the T-DOC documents that are published by the T-DOC engine. As in Bug tracker systems (such as Bugzilla [`www.bugzilla.org`]), T-DOC documents can be searched and filtered by means of ad-hoc keywords. These keywords are equivalent to the tags we defined in Section 4.3.1. For example, you can filter your search by `@author` (T-DOC documents are grouped regarding to the owner of the test cases) or by `@scope` (T-DOC documents are grouped regarding to the purpose of test cases).

As mentioned in Section 4.2, built-in test cases favour the execution of run-time testing [Suliman, et al. 2006]. The T-DOC engine exploits this feature and it is able to collect the results of the run-time execution of the test suite. Figure 11Errore: sorgente del riferimento non trovata shows the modules involved in this task. The two `T-Report` modules collect the results of the test cases execution. Hence, the two modules correlate these results with the run-time `HW/SW configuration` of the `execution environment` in which test cases have been executed. The output of these correlations are two reports (`Report a` and `Report b`) that document the results of the run-time testing activity. Currently, we are working on the identification of the profile information that should be collected by the `Profile Manager` module (such as log files, active processes, HW/SW capabilities, etc.), and we are implementing this third T-DOC layer to support the testing documentation of Java OSS projects.

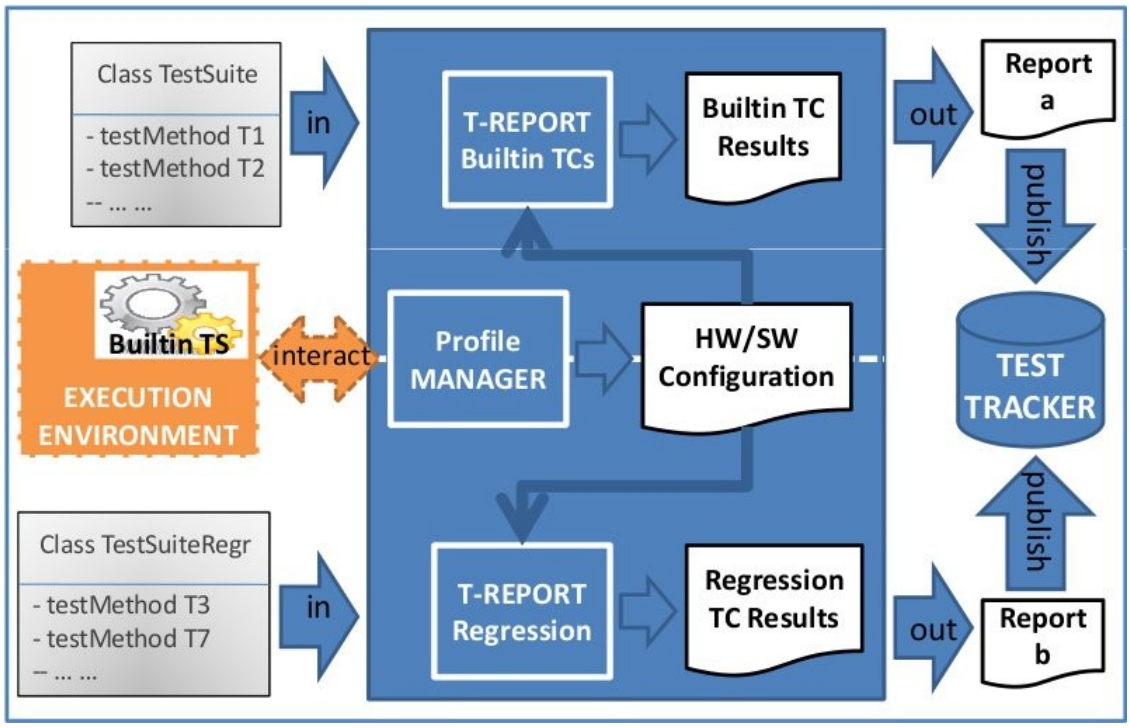

Figure 11 - Architecture of the third T-DOC layer.

4.3.4. The MACXIM Case Study

To validate the first layer of T-DOC, we applied the T-DOC method to the development of MACXIM. All the test cases, which we designed and coded for MACXIM, have been implemented following the guidelines and the documentation rules imposed by the first layer of T-DOC. The MACXIM test suite is composed of a set of unit test cases (one test case for each method of the application) and a set of acceptance test cases. For space reasons, we do not report the details of the whole test suite, but we only provide in Table 13 an excerpt of the MACXIM test plan.

Table 13 - Test Plan for MACXIM

TEST NAME	DESCRIPTION	EXPECTED RESULTS	
APPLICATION SECURITY			
LOGIN Input: username, password			
Login1	Perform the login to MacXim sending a message with a correct username and password	The login is correctly performed	
Login2	Perform the login to MacXim sending a message with an incorrect username and password	The system must return an error message	
LOGOUT Input: -			
Logout1	After Login1 test, check if the logout is properly performed	The logout is properly performed	

PROJECT MANAGEMENT			
PROJECT UPLOAD Input: project name, version, [release], [revision], repo. type, url repo, [use.rname], [password]			
ProjectUpload1	Upload project with the following parameters: project1, 1, svn, http://qualipso.dscpi.uninsubria.it/svn/svntest/ , svntest, svntest	The project is correctly uploaded	
ProjectUpload2	Upload the same project as ProjectUpload1	The system must return an error message	
GET PROJECT LIST Input: -			
GetProjectList1	After ProjectUpload1 test, check if the project is included in the list	The project is included in the list	
GET PROJECT METADATA Input: project name, version, [release], [revision]			
GetProjectMetadata1	After ProjectUpload1 test, check if metadata are correct	The project metadata are correct	
DELETE PROJECT Input: project name, version, [release], [revision]			
DeleteProject1	After ProjectUpload1 test, delete a project with the following parameters: project1, 1	The project is deleted	
DeleteProject2	After ProjectUpload1 test, delete a project with the following parameters: project1 (without specifying release number)	The system must return an error message	
PROJECT ANALYSIS all tests will be executed on the application: http://qualipso.dscpi.uninsubria.it/svn/svntest/analysis/ , svntest, svntest			
GET ALL APPLICATION LEVEL METRICS			
ApplicationMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
ApplicationMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
ApplicationMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL PACKAGE LEVEL METRICS			
PackageMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
PackageMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
PackageMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL CLASS LEVEL METRICS			
ClassMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
ClassMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
ClassMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	
GET ALL METHOD LEVEL METRICS			

MethodMetrics1	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are the same of the expected values	Values returned and expected are the same	
MethodMetrics2	Get all values (total, min, max, std-dev, median, avg) for each metric and check that no values are returned when not expected	No values are returned when not expected	
MethodMetrics3	Get all values (total, min, max, std-dev, median, avg) for each metric and check that the returned values are different from some wrong values	Returned values are different from some wrong values	

In total, 84 test cases have been implemented (47 for unit test and 37 for acceptance test). Figure 12 shows an example of a test case that follows the tag conventions defined in T-DOC. In Appendix B, we report the T-DOC documentation automatically generated for the MACXIM case study.

```
1   /**
2   *Test check if parseGetProjectList() returns exactly the project uploaded
3   * parsing the request in WebService mode or Local mode
4   *
5   *@succeedIf parseGetProjectList() returns exactly the project uploaded
6   * parsing the request in WebService mode
7   *@failIf parseGetProjectList() doesn't return exactly the project uploaded
8   * parsing the request in WebService mode
9   *@succeedIf parseGetProjectList() returns exactly the project uploaded
10  * parsing the request in Local mode
11  *@failIf parseGetProjectList() doesn't return exactly the project uploaded
12  * parsing the request in Local mode
13  *@qualityAttribute functionality
14  *@scope system
15  *@author Jacopo Emoroso
16  *@version V1.1 - 22/09/09 - 16:00
17  *@see org.uninsubria.macxim.protocol.interpreter.parser.response.MessageParser
18  */
19  @Test
20  public void testGetProjectsList() {
21
22    String[] projectList =null;
23    try {
24      projectList =(String[]) testInit.getProjectsList(AllTests.isWebService);
25    } catch (Exception e) {
26      System.out.print("Error while parsing the message");
27      e.printStackTrace();
28    }
29    assertNotNull(projectList);
30    assertTrue(Arrays.binarySearch(projectList, AllTests.uploadProjectId)==1);
31  }
```

Figure 12 - A sample MACXIM test case with T-DOC comments.

The benefits of using the T-DOC support during the MACXIM development were real:

1) the automatic generation of the testing documentation was very simple, and the documentation was always up-to-date;

2) both developers and testers had the possibility of accessing every time

they needed the testing documentation, to understand whether MACXIM contains faults or misbehaviours that need special attention;

3) communication among developers and testers was actually sped up by means of the always available documentation.

These considerations are supported both by the perceptions that had developers and testers during the development of MACXIM after the introduction of the T-DOC framework (22nd of July 2009), and also by the trend of discovered (and then fixed) faults in MACXIM, as shown in Figure 13. Figure 13 focuses on the metrics implemented in MACXIM that we were able to detect as faulty metrics by means of testing activities. The graph clearly shows that the availability of the testing documentation (the blue arrow indicates the adoption of T-DOC) simplified the process of detecting the faulty metrics, accelerated the process of correcting the faulty one, and finally favoured the process of implementing new metrics. After the adoption of the T-DOC framework (as the blue arrow shows), the number of faulty metrics decreased from 10 to 4 (-60%) in a few days, the number of corrected metrics increased inversely to the faulty metrics and the number of new metrics increased from 22 to 28 (+28%).

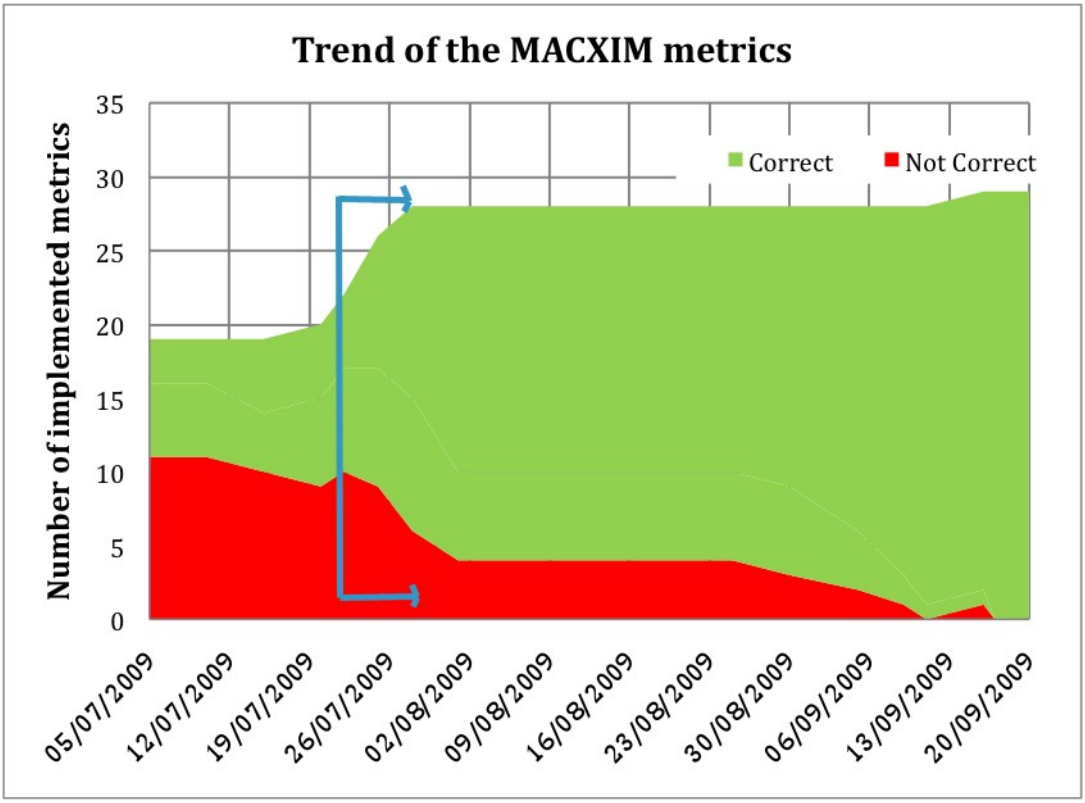

Figure 13 - Correct / Incorrect MACXIM metrics discovered by test.

4.4. Final remarks

We have outlined the T-DOC framework in this section. The T-DOC objective is to support a team of OSS developers in creating test documentation that will enhance OSS trustworthiness. It does so in several ways:

(1) T-DOC introduces a set of new tags similar to Java Doc tags to be included in testing code that will be used to generate automatic documentation;

(2) T-DOC analyses the source code to suggest integration and regression tests;

(3) T-DOC supports archiving of testing documents in central repositories.

The T-DOC framework development is still under way but we could apply T-DOC two case studies to evaluate its applicability and real benefits. Firstly, we utilized it in OSS RealEstate Java to prove the concepts underlying T-DOC. Subsequently, T-DOC was utilized in a real project involving several developers during the development of a tool for metrics collection in Java code.

T-DOC benefits we perceived both qualitatively and quantitatively. According to developers, testing documentation was easily generated; information regarding the tests was always available and up-to-date; and communication among developers was sped up. Quantitative improvements were observed as well: the number of faulty metrics decreased since the adoption of T-DOC and the number of new metrics correctly implemented increased.

We believe T-DOC addresses many issues associated to OSS testing documentation. It does not only support the document production but also drives the testing activity by suggesting the development of integration and regression tests and saving the documents created. By making the test documents available in repositories a stakeholder (a software company, a developer or an end-user) will have subsidies to assess OSS trustworthiness.

5. Conclusions

Open Source Software (OSS) products do not usually follow the traditional software engineering development paradigms found in textbooks. Specifically, testing activities in OSS development may be quite different from those carried out in Closed Source Software (CSS) development, also due to the fact that OSS processes often seem to be less structured than CSS ones. Since testing may require a good deal of resources in OSS, it is necessary to have ways for assessing and improving OSS testing processes.

In this document, we evaluated the coverage with respect to structural testing criteria provided by OSS test suites. Control- and data-flow based—all-Nodes, all-Edges, all-Uses, and all-Potential-Uses—criteria were utilized. Eight OSS projects were analyzed, namely, HSQLDB, HTTPUnit, JasperReports, JMeter, JUnit, Log4J, PMD and Velocity. The coverage data obtained reveals that in general test suites need improvement. Five analyzed OSS projects obtained code coverage below 40%; two obtained coverage data around 50%; and only one OSS project obtained coverage above 70%, which is recommended for CSS. Results along the same lines were obtained using similar control-flow metrics collected using a different tool and Aspect-oriented programming techniques.

One possible explanation of this situation resides in the very nature of testing in OSS projects. There is an assumption that OSS is tested by using it in actual settings. This point is backed in part by Raymond's argument that "Given enough eyeballs, all bugs are shallow" [Raymond 2001]. However, the negation of the first part of this argument may imply that bugs will go under whenever enough eye balls are missing.

Our assessment of the test suites of OSS projects indicates that more systematic testing is needed. Other evidence is given by our study of Busybox. The augmentation of its test suite with acceptance tests and the use of regression testing caused the software to fail and revealed three new errors. One way to overcome this situation may be to assess the code of the OSS while it is in use by regular users and to register this use as test cases. Continuous [Saff and Ernst 2004] and evolutionary [Santelices, et al. 2008] testing are techniques that may help to achieve such goal and should be investigated.

The evidences obtained with the assessments described in this document suggest that the software testing process in many OSS projects is not mature enough. To achieve a systematic evaluation of the OSS testing process and to provide a program to improve it, the Open-source Software Testing Maturity Model (OSS-TMM) was proposed.

OSS-TMM provides guidance to identify the "Best Testing Process" (*BTP*) tailored to the application of the OSS and to assess its "Available Testing Process" (*ATP*). The compliance of the *ATP* with respect to the *BTP* gives the maturity level of the testing process. OSS-TMM was utilized to analyze in detail

two real-life projects, namely, BusyBox and Apache HTTP. In addition, four more representative OSS projects were assessed with OSS-TMM in order to correlate their maturity levels with their bug rates to comprehend whether a higher maturity of the testing process directly means a higher product quality. OSS-TMM can be easily applied either to small or to large OSS projects, but the correlation between the level of maturity and bug rates was verified in part.

However, an immature OSS project may have a low bug rate (number of bugs divided by the product size in KLOC) because of its limited number of features. As shown in the BusyBox study the current test suite was unable to reveal three errors only revealed by the improvement of the testing process. Thus the current bug rate was low due to the testing process immaturity. OSS-TMM makes this lack of maturity explicit.

The code coverage might have a role in fine-tuning the correlation between the maturity levels of an OSS project and its perceived trustworthiness. A low bug rate may be an inadequate quality metric for OSS projects with immature testing process and with low coverage test suites.

Our assessment of the OSS testing documentation revealed a disturbing situation: Low effort is directed in OSS projects towards developing testing documentation. One possible explanation is because documentation (and testing documentation in particular) is perceived as an unrewarding and less reputable activity in OSS communities. A direct implication is that one (a software company, a developer or an end-user) does not have ways to assess how testing was conducted and, as result, the OSS trustworthiness.

In this document, the T-DOC framework is introduced to address some of the issues associated to testing documentation: (1) T-DOC introduces a set of new tags similar to Java Doc tags to be included in testing code that will be used to generate automatic documentation; (2) T-DOC analyses the source code to suggest integration and regression tests; (3) T-DOC supports archiving of testing documents in central repositories. We believe T-DOC will have a beneficial impact in testing of OSS. Not only by providing mechanisms to automatic creation and archival of testing documentation but also by guiding the testing activity. Our assessments (e.g., the BusyBox study) indicate that integration and regression testing have a pivotal role in improving the OSS testing activity and as a consequence in the OSS trustworthiness.

The T-DOC development is still under way and should continue during the next steps of Working Package 5.4. Nevertheless, an example (OSS RealEstate Java) was presented in this document to show the applicability and real benefits of T-DOC.

References

[1] AJDT: AspectJ Development Tools. Web published: www.eclipse.org/ajdt/. Accessed: March 2010.

[2] Arkin, B., S. Stender, and G. McGraw. "Software penetration testing." *IEEE Security and Privacy*, 2005.

[3] Arisholm, E., L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Software Eng*., 30(8):491-506, 2004.

[4] Beizer, B. *Software Testing Techniques.* Van Nostrand Reinhold Company, 1990.

[5] Beydeda, S. "Research in testing COTS components -- built-in testing approaches." *Proceedings of the ACM/IEEE International Conference on Computer Systems and Applications (AICCSA).* IEEE Computer Society Press, 2005. 101-104.

[6] Budd, T. A., R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs." *7th ACM Symposium on Principles of Programming Languages.* New York: ACM, 1980. 220–233.

[7] Burnstein, I., T. Suwanassart, and R. Carlson. "Developing a Testing Maturity Model for software test process evaluation and improvement." *IEEE International Test Conference (ITC).* New York: IEEE Computer Society Press, 1996. 581--589.

[8] Cornett, S. "Minimum Acceptable Code Coverage." 2009. http://www.bullseye.com/minimum.html (accessed June 30, 2009).

[9] Dannenberg, R. B., and G. W. Ernst. "Formal program verification using symbolic execution." *IEEE Transactions on Software Engineering*, 1982.

[10] Duijnhouwer, F., and C. Widdows. *Open Source Maturity Model.* 2003. www.osspartner.com (accessed January 31, 2009).

[11] EclEmma Eclipse Plugin. Web published: www.eclemma.org. Accessed: March 2010.

[12] Eclipse. *Eclipse Test & Performance Tools Platform Project.* 2009. www.eclipse.org/tptp/ (accessed June 30, 2009).

[13] Emam, K. E. *Spice: The Theory and Practice of Software Process Improvement and Capability dEtermination.* New York: IEEE Computer Society Press, 1997.

[14] Ericson, T., A. Subotic, and S. Ursing. "TIM - a Test Improvement Model." *International Journal on Software Testing, Verification and Reliability 7 (4)*, 1997: 229--246.

[15] Ernst, M. D., J. Cockrell, W. G. Griswold, and D. Notkin. "Dynamically discovering likely program invariants to support program evolution." *IEEE Transactions on Software Engineering*, 2001.

[16] Frankl, P. G., and O. Iakounenko. "Further Empirical Studies of Test

Effictiveness." *Proceedings of the ACM SIGSOFT Foundations of Software Engineering Conference.* 1998.

[17] Frankl, P., and S. Weiss. "An experimental comparison of the effectiveness of branch testing and data flow testing." *IEEE Transaction on Software Engineering (19):8*, 1993: 774-787.

[18] Herbsleb, J., D. Zubrow, D. Goldenson, W. Hayes, and M. Paulk. "Software quality and the Capability Maturity Model." *Communications of the ACM 40 (6)* , 1997: 30--40.

[19] Ho, H., S. Elbaum, and G. Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." *Empirical Software Engineering (10)*, 2005: 405–435.

[20] Howard, M. "A process for performing security code reviews." *IEEE Security and Privacy*, 2006, 4 ed.

[21] Hutchins, M., H. Foster, T. Goradia, and T. Ostrand. "Experiments on the Effectiveness of Dataflow- and Control flow-Based Test Adequacy Criteria." *Proceedings of the 16th International Conference on Software Engineering.* Sorento, Italy: IEEE Computer Society Press, 1994. 191-200.

[22] IEEE. *IEEE Standard for Software Unit Testing.* ANSI/IEEE Std 1008-1987, New York: IEEE Computer Society Press, 1987.

[23] ISO1. *Information technology - software product evaluation - part 1: General overview.* ISO/IEC 14598-1, International Organization for Standardization, 2001.

[24] ISO2. *Information technology process assessment - part 1: Concepts and vocabulary.* ISO/IEC 15504-1, International Organization for Standardization, 2004.

[25] ISO3. *Software engineering - product quality - part 1: Quality model.* ISO/IEC 9126-1, International Organization for Standardization, 2001.

[26] King, G., Y. Wang, and H. Wickburg. "A Method for Built-in Tests in Component-based Software Maintenance." *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society Press, 1999. 186--192.

[27] Koomen, T., and M. Pol. *Test Process Improvement: a practical step-by-step guide to structured testing.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[28] Maldonado, J. C. *Potential-uses criteria: A contribution to the structural. Ph. D Thesis.* Campinas, SP, Brazil: State University of Campinas, 1991.

[29] Mao, C., Y. Lu, and J. Zhang. "Regression testing for component-based software via built-in test design." *Proceedings of the ACM Symposium on Applied Computing (SAC).* 2007. 1416--1421.

[30] Mockus, A., R. T. Fielding, and J. Herbsleb. "A case study of OSS development: the apache server." *Proceedings of the International Conference on Software Engineering (ICSE).* 2000. 263--272.

[31] Morell, L. J. "A theory of fault-based testing." *IEEE Transactions on Software Engineering (TSE) 16 (8)*, 1990: 844--857.

[32] Myers, Glenford J. *The Art of Software Testing.* Wiley, 2004.

[33] NCSU. *RealEstate Example.* 2009. http://agile.csc.ncsu.edu/SEMaterials/realestate/ (accessed August 2009, 1).

[34] O'Reilly, T. "Lessons from Open-Source Software development." *Communications of the ACM 42 (4).*

[35] Orso, A., M. J. Harrold, D. S. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. "Using Component Metacontent to Support the Regression Testing of Component-Based Software." *Proceedings of the IEEE International Conference on Software Manintenance (ICSM).* IEEE Computer Press, 2001. 716--725.

[36] Ostrand, T. J., and M. J. Balcer. "The category-partition method for specifying and generating functional tests." *Communications of the ACM 31 (6)*, 1988: 676--686.

[37] Pezzè, M., and M. Young. *Software Testing And Analysis. Process, Principles, and Techniques.* Wiley, 2007.

[38] Pfleeger, S. L. *Software Engineering: Theory and Practice.* Prentice-Hall, 2009.

[39] Pretschner, A., et al. "One evaluation of model-based testing and its automation." *Proceedings of the 27th International Conference on Software Engineering (ICSE).* New York: IEEE Computer Society Press, 2005. 392--401.

[40] Qualipso. *Definition of standard test approaches, test suites, and benchmarks of Open Source Software.* February 25, 2009. http://www.qualipso.eu/node/129 (accessed July 24, 2009).

[41] QualiPSo1. *How European software industry perceives OSS trustworthiness and what are the specific criteria to establish trust in OSS.* 2009. http://www.qualipso.eu/node/45 (accessed June 30, 2009).

[42] QualiPSo2. *Analysis of relevant open source projects.* 2009. http://www.qualipso.eu/node/84 (accessed June 30, 2009).

[43] Rapps, S., and E. J. Weyuker. "Selecting software test data using data flow." *Transaction on Software Engineering, vol. 11, no. 4*, 1985: 367--375.

[44] Raymond, E. S. *The Cathedral and the Bazaar.* O'Reilly Media, 2001.

[45] RTCA. *RTCA Software Considerations in Airborne Systems and Equipment Certification Radio Technical Commission for Aeronautics.* RTCA/DO-178B , RTCA, 1992.

[46] Saff, D., and M. D. Ernst. "An experimental evaluation of continuous testing during development." *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* Boston, MA, 2004. 76--85.

[47] Santelices, R. A., P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. "Test-suite augmentation for evolving software." *IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2008.

218--227.

[48] Sinha, S., and M. J. Harrold. "Analysis of programs with exception-handling." *ICSM'98 -- International Conference on Software.* Bethesda, 1998. 348--357.

[49] Suliman, D., et al. "The MORABIT Approach to Runtime Component Testing." *Proceedings of the International Computer Software and Applications Conference (COMPSAC).* IEEE Computer Society Press, 2006. 171--176.

[50] Taibi, D., L. Lavazza, and S. Morasca. "OpenBQR: a framework for the assessment of OSS." *International Journal on Open Source Development, Adoption and Innovation*, 2007: 173--186.

[51] Tosi, Davide, Davide Taibi, and Sandro Morasca. "Improving the Testing Process of Open Source Software Systems." *Journal of Software and Systems*, 2009: to appear.

[52] Vincenzi, A. M. R., J. C. Maldonado, W. E. Wong, and M. E. Delamaro. "Coverage Testing of Java Programs and Components." *Journal of Science of Computer Programming, vol. 56*, 2005: 211-230.

APPENDIX A – OSS-TMM CHECKLIST

The appendix is presented as a checklist where each issue is represented as a set of questions. Each question stresses a specific sub-issue that can (or cannot) characterize the OSS project under concern. Each question is annotated with a set of predefined answers and remarks about testing guidelines. Questions, answers, and guidelines are formulated to limit subjectivity.

I1 – Visibility

I1.1 Is the source code available via Versioning Systems?

a1: yes, the whole project is managed via SVN/CVS

a2: only some features are managed via SVN/CVS

a3: no, the project is not managed via SVN/CVS

I1.2 Is the project structured in folders containing sources, binaries, libraries, docs a1: yes, the whole project is well structured

a2: the project is not completely structured

a3: no, the project is not structured

I1.3 Is information about releases (date, number, change log) visible?

a1: yes, all the info are provided

a2: only some info are provided

a3: no, no info is provided

I1.4 Is information about code revisions (author, date, number, description) visible?

a1: yes, all the info are provided

a2: only some info are provided

a3: no, no info is provided

I1.5 Are security issues meaningful for the product?

a1: yes, private data are manipulated (bank accounts)

a2: only sensible data are manipulated (name, surname)

a3: no, no data are implicitly or explicitly manipulated

I1.6 Are the scripts of the test cases open source?

a1: yes, all the scripts are open source to the community

a2: only some scripts are open source

a3: no, no scripts are open source to the community

I1.7 Is the source code released under several OSI licenses (see `www.opensource.org/licenses/` for an exhaustive list of licenses approved by the Open Source Initiative (OSI))?

a1: yes, popular licenses are used

a2: not the whole code is released under popular licenses

a3: no, non popular licenses are used

I1.8 Are log files about system executions available?

a1: yes, log files are available

a2: no, log files are not available

I2 - System Analysis and Product Design Activities

I2.1 Is a project plan/roadmap available?

a1: yes, the project plan/roadmap is available

a2: no, the project plan/roadmap is not available

I2.2 Is a risk analysis available?

a1: yes, the risk analysis is available

a2: no, the risk analysis is not available

I2.3 Is a requirements analysis available?

a1: yes, the requirements analysis is available

a2: no, the requirements analysis is not available

I2.4 Is a goal analysis available?

a1: yes, the goal analysis is available

a2: no, the goal analysis is not available

I2.5 Are system designs available (UML or other notations)?

a1: yes, use case, class and sequence diagrams are available

a2: partially, only some diagrams are available

a3: no, system designs are not available

I2.6 Are standard protocols or patterns identified?

a1: yes, standard protocols/patterns are identified

a2: no, standard protocols/patterns are not identified

I2.7 Are coding standards and conventions identified?

a1: yes, coding standards are identified

a2: no, coding standards are not identified

I2.8 Are SLAs or performance requirements meaningful?

a1: yes, real-time constraints

a2: normal constraints (memory, bandwidth, latency)

a3: no, no constraints are implicitly or explicitly visible

I2.9 Does the system follow a specific architectural style (e.g., SOA, peer-to-peer, etc)?

a1: yes, the system follows a specific architecture

a2: no, the system does not follow a specific architecture

I2.10 Is the system developed with a GUI?

a1: yes, the system has a GUI

a2: no, the system does not have a GUI

I2.11 Does the product use external libraries/plugins?

a1: yes, the product uses external libraries/plugins

a2: no, the product does not use the external libraries/plugins

I3 - Development Process

I3.1 Is a specific development process used (e.g., waterfall, XP, continuous building)?

a1: yes, a specific development process is followed

a2: no, a specific development process is not followed

I3.2 Are developers/contributors structured in teams?

a1: yes, different teams exist

a2: no, developers are unstructured

I3.3 Does the system provide a sand box environment?

a1: yes, a sandbox environment is available

a2: no, a sandbox environment is not available

I3.4 Is a specific IDE used/recommended?

a1: yes, a specific IDE is used/recommended

a2: no, a specific IDE is not used/recommended

I3.5 Is a testing platform used/recommended?

a1: yes, a specific testing platform is used/recommended

a2: no, a specific testing platform is not used/recommended

I3.6 Is a bug tracking system available?

a1: yes, the bug tracking system is available

a2: no, the bug tracking system is not available

I4 - System Growth and Community Creativity

I4.1 Is the number of code changes per release >500?

a1: yes, the number of code changes is >500

a2: no, the number of code changes is <=500

I4.2 Is the number of developers/contributors >100? (small community size <10; medium size <100; big size >100)

a1: yes, the number of developers/contributors is >100

a2: no, the number of developers/contributors is <=100

I4.3 Does the system have different releases/system's updates?

a1: yes, the number of releases/updates is provided

a2: no, the number of releases/updates is not provided

I4.4 Is the number of open bugs/fixed bugs/... available?

a1: yes, statistics about bugs are provided

a2: no, statistics about bugs are not provided

I4.5 Is the frequency of changes/updates/bug time solving recognizable? (to evaluate whether the project is still alive)

a1: yes, the frequency of code changes is recognizable

a2: no, the frequency of code changes is not recognizable

I5 - Documentation and Dissemination

I5.1 Is a system-level documentation available?

a1: yes, the system-level documentation is available

a2: no, the system-level documentation is not available

I5.2 Is a library-level documentation available?

a1: yes, the library-level documentation is available

a2: no, the library-level documentation is not available

I5.3 Is a feature-level documentation available?

a1: yes, the feature-level documentation is available

a2: no, the feature-level documentation is not available

I5.4 Is a user manual available?

a1: yes, the user manual is available

a2: no, the user manual is not available

I5.5 Are bugs reports available?

a1: yes, bug reports are available

a2: no, bug reports are not available

I5.6 Is code documentation available (javadoc, etc)?

a1: yes, the code documentation is available

a2: no, the code documentation is available

I5.7 Are documents disseminated via unstructured channels (mailing lists, forums, etc)?

a1: yes, docs are disseminated via unstructured channels

a2: no, docs are not disseminated via unstructured channels

I5.8 Are Installation requirements documented?

a1: yes, installation requirements are available

a2: no, installation requirements are not available

I5.9 Are test-plan/test-design/test-results documents available?

a1: yes, documentation about testing is available

a2: no, documentation about testing is not available

Overall Testing Remarks

I1.1 to I1.4 are TRUE: unit testing, integration testing, and regression testing activities are facilitated

I1.5 and I1.6 are TRUE: security testing (formal testing for the functions that manipulate private/sensible data, penetration tests, dependencies tests, risk-based security tests) is suggested

I1.7 is TRUE: check the compatibility among the different licenses adopted by the project

I1.8 is TRUE: dynamic analysis techniques are applicable

I2.1 to I2.4 are TRUE: exploit requirements to design oracles and test cases for black box testing activities (such as category partition, catalogs, hw/sw

requirements testing, acceptance testing)

I2.5 is TRUE: exploit models to perform model-based testing

I2.6 is TRUE: conformance testing to verify the protocol behaviour

I2.7 is TRUE: style check and inspection to verify the conventions

I2.8 is TRUE: apply performance testing (such as load testing, stress testing, endurance testing)

I2.9 is TRUE: select the testing techniques specialized for the chosen architectural style (for example, service-oriented architectures prefer on-line testing techniques)

I2.10 is TRUE: apply capture\&replay and usability testing

I2.11 is TRUE: check compliance through versioning compatibility checks and installation testing activities

I3.1 is TRUE: verify whether the testing process is in line with the chosen development process

I3.2 is FALSE and I3.3 is TRUE: increase the integration and regression testing activities and use the sand box as the testing environment

I3.4 is TRUE: make the most of the testing potentialities offered by the chosen IDE

I3.5 is FALSE: select a testing framework to support and automatize the testing process

I3.6 is TRUE: apply fault-based testing techniques

I4: proportionally to the size of the community and the vitality of the project, improve: the integration and regression testing activity; the testing automation; the sharing of testing knowledge to increase the reusability of test suites; the

documentation of test-strategy/tests-results; the monitoring of testing activities

I5.1 to I5.8 are TRUE: exploit the documentation to simplify acceptance/system testing, usability testing, installation testing

I5.9 is FALSE: provide testing documentation through "test management tools" (such as TestLink, qaManager, etc.) that automatize/simplify the generation of reports

APPENDIX B – MACXIM T-DOC DOCUMENTATION

Class ApplicationLevelMetricsTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.ApplicationLevelMetricsTest
public class ApplicationLevelMetricsTest extends java.lang.Object
```

Test cases relative to metrics with Application granularity level.

Author:
Massimiliano Bosetti, Vincenzo Pandico, Jacopo Emoroso
Version:
1.1 - 24/07/09 - 15:31

Field Summary

`private java.lang.String regex`
String used for catching regular expressions.
`private static java.lang.String response`
`private stub`
`static org.uninsubria.macxim.ws.MacXimProxy` The stub object, entry point for Macxim processing.

Constructor Summary

`ApplicationLevelMetricsTest()`

Method Summary

`static void init()`
Initialize the test case setting variables and executing a login to MacXim calling the method `executeAllMetrics`.
`void testExecuteMetricsCBO()`
Test try to execute the metric CBO and check the values for total, max, min, stddev, median, avg.
`void testExecuteMetricsCommentLinesPerClass()`
Test try to execute the metric Comment Lines per Class and check the values for total, max, min, std-dev, median, avg.
`void testExecuteMetricsEloc()`
Test try to execute the metric eLOC per Class and check the values for total, max, min, std-dev, median, avg.
`void testExecuteMetricsLCOM()`

Test try to execute the metric LCOM and check the values for total, max, min, stddev, median, avg.

`void testExecuteMetricsMcCabe()`

Test try to execute the metric McCabe and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsNumAttributesPerClass()`

Test try to execute the metric Number of Attributes per Class and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsNumClasses()`

Test try to execute the metric Number Classes and check the value for total.

`void testExecuteMetricsNumClassesWithDefinedAttributes()`

Test try to execute the metric Number of Classes with Defined Attributes and check the value for total.

`void testExecuteMetricsNumClassesWithDefinedMethods()`

Test try to execute the metric Number of Classes with Defined Methods and check the value for total.

`void testExecuteMetricsNumInterfacesPerClass()`

Test try to execute the metric Number of Interfaces per Class and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsNumMethods()`

Test try to execute the metric Number of Methods and check the value for total.

`void testExecuteMetricsNumMethodsPerClass()`

Test try to execute the metric Number of Methods per Class and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsNumMethodsPerInterface()`

Test try to execute the metric Number of Methods per Interface and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsNumPackages()`

Test try to execute the metric Number of Packages and check the value for total.

`void testExecuteMetricsNumParametersPerMethod()`

Test try to execute the metric Number of Parameters per Method and check the values for total, max, min, std-dev, median, avg.

`void testExecuteMetricsRFC()`

Test try to execute the metric RFC and check the values for total, max, min, stddev, median, avg.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

regex

`private java.lang.String` **`regex`** String used for catching regular expressions.

response

`private static java.lang.String` **`response`**

stub

`private static org.uninsubria.macxim.ws.MacXimProxy` **`stub`** The stub object, entry point for Macxim processing.

Constructor Detail

ApplicationLevelMetricsTest

`public` **`ApplicationLevelMetricsTest`**`()`

Method Detail

init

`public static void` **`init`**`()`
`throws java.lang.Exception`
Initialize the test case setting variables and executing a login to MacXim calling the method `executeAllMetrics`. It is executed just one time at the beginning of test case.
Throws:
`java.lang.Exception` - exception

testExecuteMetricsCBO

`public void` **`testExecuteMetricsCBO`**`()`
Test try to execute the metric CBO and check the values for total, max, min, std-dev, median, avg. **See Also:**
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsCommentLinesPerClass

`public void` **`testExecuteMetricsCommentLinesPerClass`**`()`

Test try to execute the metric Comment Lines per Class and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsEloc

`public void testExecuteMetricsEloc()`
Test try to execute the metric eLOC per Class and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsLCOM

`public void testExecuteMetricsLCOM()`
Test try to execute the metric LCOM and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:

Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsMcCabe

`public void testExecuteMetricsMcCabe()`
Test try to execute the metric McCabe and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:

V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0 **Test Succeed If:**
the values are initialized and each value is > 0

testExecuteMetricsNumAttributesPerClass

`public void testExecuteMetricsNumAttributesPerClass()`
Test try to execute the metric Number of Attributes per Class and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:

Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumClasses

`public void testExecuteMetricsNumClasses()`
Test try to execute the metric Number Classes and check the value for total. **See Also:**
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:

Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0 **Test Succeed If:**
the values are initialized and each value is > 0

testExecuteMetricsNumClassesWithDefinedAttributes

`public void testExecuteMetricsNumClassesWithDefinedAttributes()`
Test try to execute the metric Number of Classes with Defined Attributes and check the value for total.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumClassesWithDefinedMethods

`public void testExecuteMetricsNumClassesWithDefinedMethods()`
Test try to execute the metric Number of Classes with Defined Methods and check the value for total.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0

Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumInterfacesPerClass

`public void testExecuteMetricsNumInterfacesPerClass()`
Test try to execute the metric Number of Interfaces per Class and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumMethods

`public void testExecuteMetricsNumMethods()`
Test try to execute the metric Number of Methods and check the value for total. **See Also:**
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:

Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumMethodsPerClass

`public void testExecuteMetricsNumMethodsPerClass()`
Test try to execute the metric Number of Methods per Class and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumMethodsPerInterface

`public void testExecuteMetricsNumMethodsPerInterface()`
Test try to execute the metric Number of Methods per Interface and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:

Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumPackages

`public void testExecuteMetricsNumPackages()` Test try to execute the metric Number of Packages and check the value for total. **See Also:**
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsNumParametersPerMethod

`public void testExecuteMetricsNumParametersPerMethod()`
Test try to execute the metric Number of Parameters per Method and check the values for total, max, min, std-dev, median, avg.
See Also:
`java.util.regex.Pattern.matcher(CharSequence arg0)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0
Test Succeed If:
the values are initialized and each value is > 0

testExecuteMetricsRFC

`public void` **`testExecuteMetricsRFC()`** Test try to execute the metric RFC and check the values for total, max, min, std-dev, median, avg. **See Also:**
`java.util.regex.Pattern.matcher(CharSequence arg0)`
Author:

Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:31
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the values are not initialized properly or at less one value is < 0 **Test Succeed If:**
the values are initialized and each value is > 0

Overview Package Class **Use Tree Index Help**
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Overview Package Class **Use Tree Index Help**
PREV CLASS **NEXT CLASS FRAMES NO FRAMES All Classes** SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
org.uninsubria.macxim.mw.database.test

Class DBAdministrationTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.database.test.DBAdministrationTest
public class DBAdministrationTest extends java.lang.Object
```
Test cases relative to database administration functionalities. **Author:**
Massimiliano Bosetti, Jacopo Emoroso

Constructor Summary

`DBAdministrationTest()`

Method Summary

`void testGetDBURI()`
Test try to find the correct URI used for the database binding.
`void testGetPassword()`
Test try to find the password for the access to database.
`void testGetUsername()`
Test try to find the user admin for the access to database.
`static void testInitialize()`
Initialization of Test DBAdministration, that try to run database administration functionalities.
`static void testShutdown()`

Closing of Test DBAdministration.
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DBAdministrationTest

```
public DBAdministrationTest()
```

Method Detail

testGetDBURI

```
public void testGetDBURI()
```
Test try to find the correct URI used for the database binding.
See Also:
`org.uninsubria.macxim.mw.database.DBAdministration.getDBURI()`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the URI used for the database binding is different from "xmldb:exist:///db"
Test Succeed If:
the URI used for the database binding is "xmldb:exist:///db"

testGetPassword

```
public void testGetPassword()
```
Test try to find the password for the access to database. **See Also:**
`org.uninsubria.macxim.mw.database.DBAdministration.getPassword()`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:

Functionality
Test Fail If:
the password max1982 doesn't exist for the access to database
Test Succeed If:
the password max1982 exist for the access to database

testGetUsername

`public void testGetUsername()`
Test try to find the user admin for the access to database. **See Also:**
`org.uninsubria.macxim.mw.database.DBAdministration.getUsername()`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the user "admin" doesn't exist for the access to database
Test Succeed If:
the user "admin" exist for the access to database

testInitialize

`public static void testInitialize()` Initialization of Test DBAdministration, that try to run database administration functionalities.

testShutdown

`public static void testShutdown()`
Closing of Test DBAdministration.
Overview Package Class **Use Tree Index Help**
PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes** SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Overview Package Class **Use Tree Index Help**
PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes** SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD **org.uninsubria.macxim.mw.database.test**

Class DBQueryTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.database.test.DBQueryTest
```
`public class DBQueryTest extends java.lang.Object`
Test cases relative to database query functionalities. **Author:**
Massimiliano Bosetti, Jacopo Emoroso

Field Summary

`private dbq static org.uninsubria.macxim.mw.database.DBQuery`

Constructor Summary

`DBQueryTest()`

Method Summary

`static void initialize()`
Initialization of Test DBQueryTest, that try to run database query functionalities.
`static void shutdown()`
Closing of Test DBQueryTest.
`void testExecuteQueryCode()`
Test try to execute a query code with some parameters.
`void testExecuteQueryString()`
Test try to execute a database query without parameters.
`void testExecuteQueryStringMap()`
Test try to execute a database query with parameters.
`void testGetQuery()`
Test try to get the correct query by its name passed as parameter.
`void testGetQueryList()`
Test try to get the correct query list from the database.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

dbq

`private static org.uninsubria.macxim.mw.database.DBQuery dbq`

Constructor Detail

DBQueryTest

`public DBQueryTest()`

Method Detail

initialize

`public static void initialize()` Initialization of Test DBQueryTest, that try to run database query functionalities.

shutdown

`public static void shutdown()` Closing of Test DBQueryTest.

testExecuteQueryCode

`public void testExecuteQueryCode()`
Test try to execute a query code with some parameters. **See Also:**
`org.uninsubria.macxim.mw.database.DBQuery.executeQueryCode(String queryCode, Map parameters)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the execution of the query code doesn't return the correct result
Test Succeed If:
the execution of the query code returns the correct result

testExecuteQueryString

`public void testExecuteQueryString()`
Test try to execute a database query without parameters. **See Also:**
`org.uninsubria.macxim.mw.database.DBQuery.executeQuery(String queryName)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the query "queryTest.xql" doesn't return the correct result
Test Succeed If:
the query "queryTest.xql" returns the correct result

testExecuteQueryStringMap

`public void testExecuteQueryStringMap()`
Test try to execute a database query with parameters. **See Also:**
`org.uninsubria.macxim.mw.database.DBQuery.executeQuery(String queryName, Map parameters)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the query "parametersQueryTest.xql" doesn't return the correct result
Test Succeed If:
the query "parametersQueryTest.xql" returns the correct result

testGetQuery

`public void testGetQuery()` Test try to get the correct query by its name passed as parameter. **See Also:**
`org.uninsubria.macxim.mw.database.DBQuery.getQuery(String queryName)`
Author:

Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the body of query returned isn't equals to "queryTest.xql" **Test Succeed If:**
the body of query returned is equals to "queryTest.xql"

testGetQueryList

`public void testGetQueryList()`
Test try to get the correct query list from the database. **See Also:**
`org.uninsubria.macxim.mw.database.DBQuery.getQueryList()`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the query list doesn't contain the correct elements
Test Succeed If:
the query list contains the correct elements

org.uninsubria.macxim.mw.database.test

Class DBUtilitiesTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.database.test.DBUtilitiesTest
```

public class **DBUtilitiesTest** extends java.lang.Object Test cases relative to database utilities.

Author:
Massimiliano Bosetti, Jacopo Emoroso

Field Summary

private
static org.xmldb.api.base.Collection testCollection

Constructor Summary

DBUtilitiesTest()

Method Summary

static void **createTestCollection**()
Initialization of Test DBUtilitiesTest
static void **deleteTestCollection**()
Closing of Test DBUtilitiesTest
void **testGetCollection**()
Test try to find the specified database collection into the parent collection. void
testGetDBProperty()
Test looks for a property, passed as parameter, in the file macxim.ini void
testGetStringQuery()
Test try to get the body of the query stored into a resource void **testStoreResource**()
Test try to store a resource in a database collection passed as parameter

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

testCollection private static org.xmldb.api.base.Collection **testCollection**

Constructor Detail

DBUtilitiesTest public **DBUtilitiesTest**()

Method Detail

createTestCollection

`public static void createTestCollection()` Initialization of Test DBUtilitiesTest

deleteTestCollection

`public static void deleteTestCollection()` Closing of Test DBUtilitiesTest

testGetCollection

`public void testGetCollection()` Test try to find the specified database collection into the parent collection.

See Also:
`org.uninsubria.macxim.mw.database.DBUtilities.getCollection(String parentPath, String collectionName, boolean create)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
unit
Quality Attribute:
functionality
Test Fail If:
the specified database collection isn't found
Test Succeed If:
the specified database collection is found

testGetDBProperty

`public void testGetDBProperty()` Test looks for a property, passed as parameter, in the file macxim.ini
See Also:
`org.uninsubria.macxim.common.utilities.FileSystemUtils.getFileConfigurationProperty(String section, String propertyName)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
unit
Quality Attribute:
functionality
Test Fail If:
the property passed as parameter isn't found in the file macxim.ini
Test Succeed If:
the property passed as parameter is found in the file macxim.ini

testGetStringQuery

`public void testGetStringQuery()` Test try to get the body of the query stored into a resource

See Also:

`org.uninsubria.macxim.mw.database.DBUtilities.getStringQuery(Resource query)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
unit
Quality Attribute:
functionality
Test Fail If:
the body of the query stored in the resource "resourceTest" is different from "test resource"
Test Succeed If:
the body of the query stored in the resource "resourceTest" is equals to "test resource"

testStoreResource

`public void testStoreResource()` Test try to store a resource in a database collection passed as parameter

See Also:
`org.uninsubria.macxim.mw.database.DBUtilities.storeResource(String resourcePath, String resourceName, Collection containerCollection)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
unit
Quality Attribute:
functionality
Test Fail If:
the resource isn't stored in the database collection specified

Test Succeed If:
the resource is stored in the database collection specified

Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.uninsubria.macxim.mw.test

Class JUnitUtils

```
java.lang.Object
└ org.uninsubria.macxim.mw.test.JUnitUtils
public class JUnitUtils extends java.lang.Object
```

Constructor Summary

`JUnitUtils()`

Method Summary

`static java.lang.Object executeMethod(java.lang.Object instance, java.lang.String name, java.lang.Object[] params)`
Executes a method on an object instance.
`static java.lang.Object getField(java.lang.Object instance, java.lang.String name)`
Gets the field value from an instance.
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

JUnitUtils

`public JUnitUtils()`

Method Detail

executeMethod

`public static java.lang.Object executeMethod(java.lang.Object instance, java.lang.String name, java.lang.Object[] params) throws java.lang.Exception`

Executes a method on an object instance. The name and parameters of the method are specified. The method will be executed and the value of it returned, even if the method would have private or protected access.

Throws:
`java.lang.Exception`

getField

`public static java.lang.Object getField(java.lang.Object instance, java.lang.String name) throws java.lang.Exception` Gets the field value from an instance. The field we wish to retrieve is specified by passing the name. The value will be returned, even if the field would have private or protected access. **Throws:**
`java.lang.Exception`
Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD **org.uninsubria.macxim.mw.parser**

Class LoginLogoutTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.LoginLogoutTest
public class LoginLogoutTest extends java.lang.Object
```
Test cases relative to login and logout operations to Macxim.

Author:
Jacopo Emoroso
Version:
2.0 - 24/07/09 - 15:31

Constructor Summary

`LoginLogoutTest()`

Method Summary

`void testLoginBadFormatted()`
Test check if the Login to Macxim return a correct sToken sending a bad formatted xml message, parsing the request in WebService mode or Middleware mode.
`void testLoginNo()`

Test check if the Login to Macxim return a correct sToken sending an incorrect xml message (username: "macxim", password: "macximxx"), parsing the request in WebService mode or Middleware mode.

`void testLoginOk()`

Test check if the Login to Macxim return a correct sToken sending a correct xml message (username: "macxim", password: "macxim"), parsing the request in WebService mode or Middleware mode.

`void testLogoutBadSessionToken()`
Test try to perform the Logout to Macxim sending an incorrect xml message with a wrong session token, parsing the request in WebService mode or Middleware mode.
`void testLogoutOk()`
Test try to perform the Logout to Macxim sending a correct xml message, parsing the request in WebService mode or Middleware mode.
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

LoginLogoutTest

```
public LoginLogoutTest()
```

Method Detail

testLoginBadFormatted

```
public void testLoginBadFormatted()
```

Test check if the Login to Macxim return a correct sToken sending a bad formatted xml message, parsing the request in WebService mode or Middleware mode.

See Also:
`testInit.login(boolean isWebService, String username, String password)`

Author:
Jacopo Emoroso
Version:
V2.0 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Security
Test Fail If:
sToken isn't equal to -1, parsing the request in WebService mode, sToken isn't equal to -1, parsing the request in Middleware mode
Test Succeed If:
sToken is equal to -1, parsing the request in WebService mode, sToken is equal to -1, parsing the request in Middleware mode

testLoginNo

```
public void testLoginNo()
```

Test check if the Login to Macxim return a correct sToken sending an incorrect xml message (username: "macxim", password: "macximxx"), parsing the request in WebService mode or Middleware mode.

See Also:
`testInit.login(boolean isWebService, String username, String password)`

Author:
Jacopo Emoroso
Version:
V2.0 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Security
Test Fail If:

sToken isn't equal to -1, parsing the request in WebService mode, sToken isn't equal to -1, parsing the request in Middleware mode
Test Succeed If:
sToken is equal to -1, parsing the request in WebService mode, sToken is equal to -1, parsing the request in Middleware mode

testLoginOk

```
public void testLoginOk()
```

Test check if the Login to Macxim return a correct sToken sending a correct xml message (username: "macxim", password: "macxim"), parsing the request in WebService mode or Middleware mode.

See Also:
`testInit.login(boolean isWebService, String username, String password)`

Author:
Jacopo Emoroso
Version:
V2.0 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Security
Test Fail If:
sToken isn't greater than 0, parsing the request in WebService mode, sToken isn't greater than 0, parsing the request in Middleware mode
Test Succeed If:
sToken is greater than 0, parsing the request in WebService mode, sToken is greater than 0, parsing the request in Middleware mode

testLogoutBadSessionToken

```
public void testLogoutBadSessionToken()
```

Test try to perform the Logout to Macxim sending an incorrect xml message with a wrong session token, parsing the request in WebService mode or Middleware mode.
See Also:

`testInit.logout(boolean isWebService, long sessionToken)`

Author:
Jacopo Emoroso
Version:
V2.0 - 22/09/09 - 16:00
Test Scope:
Acceptance

Quality Attribute:
Security
Test Fail If:
testInit.logout(sToken) return true, parsing the request in WebService mode,
testInit.logout(sToken) return true, parsing the request in Middleware mode
Test Succeed If:
testInit.logout(sToken) return false, parsing the request in WebService mode,
testInit.logout(sToken) return false, parsing the request in Middleware mode

testLogoutOk

`public void testLogoutOk()`
Test try to perform the Logout to Macxim sending a correct xml message, parsing the request in WebService mode or Middleware mode.
See Also:
`testInit.logout(boolean isWebService, long sessionToken)`

Author:
Jacopo Emoroso
Version:
V2.0 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Security
Test Fail If:
testInit.logout(sToken) return false, parsing the request in WebService mode,
testInit.logout(sToken) return false, parsing the request in Middleware mode
Test Succeed If:
testInit.logout(sToken) return true, parsing the request in WebService mode,
testInit.logout(sToken) return true, parsing the request in Middleware mode

Class ProjectManagementTest

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.ProjectManagementTest
public class ProjectManagementTest extends java.lang.Object
```
Test cases relative to Macxim project management operations.

Author:
Massimiliano Bosetti, Vincenzo Pandico, Jacopo Emoroso
Version:

V1.1 - 22/09/09 - 16:00

Constructor Summary

`ProjectManagementTest()`

Method Summary

`void testGetProjectMetadata()`

Test check if the metadata of a project uploaded obtained with the method `getProjectMetadata` return the correct xml response, parsing the request in WebService mode or Middleware mode.

`void testGetProjectsList()`
Test check if the list of projects uploaded obtained with the method`getProjectsList` return the correct xml response, parsing the request in WebService mode or Middleware mode.
`void testRemoveProject()`
Test check if the delete of a project with`removeProject` method return the correct xml response, parsing the request in WebService mode or Middleware mode.
`void testUpdateProjectMetadata()`
Test check if the upload of metadata of a project with`uploadProjectMetadata` method return the correct xml response, parsing the request in WebService mode or Middleware mode.
`void testUploadProject()`
Test check if the upload of a project with`uploadProject` method return the correct xml response, parsing the request in WebService mode or Middleware mode.
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ProjectManagementTest

`public ProjectManagementTest()`

Method Detail

testGetProjectMetadata

`public void testGetProjectMetadata()`
Test check if the metadata of a project uploaded obtained with the method`getProjectMetadata` return the correct xml response, parsing the request in WebService mode or Middleware mode. **See Also:**
`org.uninsubria.macxim.mw.parser.testInit.getProjectMetadata(boolean isWebService, String projectId)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the response is an incorrect xml message, parsing the request in WebService mode, the response is an incorrect xml message, parsing the request in Middleware mode
Test Succeed If:
the response is a correct xml message, parsing the request in WebService mode, the response is a correct xml message, parsing the request in Middleware mode

testGetProjectsList

`public void testGetProjectsList()`
Test check if the list of projects uploaded obtained with the method`getProjectsList` return the correct xml response, parsing the request in WebService mode or Middleware mode. **See Also:**
`org.uninsubria.macxim.mw.parser.testInit.getProjectsList(boolean isWebService)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:

the response isn't a list with one project or idProject isn't 1, parsing the request in WebService mode, the response isn't a list with one project or idProject isn't 1, parsing the request in Middleware mode

Test Succeed If:
the response is a list with one project and idProject is 1, parsing the request in WebService mode, the response is a list with one project and idProject is 1, parsing the request in Middleware mode

testRemoveProject

`public void testRemoveProject()`
Test check if the delete of a project with`removeProject` method return the correct xml response, parsing the request in WebService mode or Middleware mode.
See Also:

`org.uninsubria.macxim.mw.parser.testInit.removeProject(boolean isWebService, String projectId)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the response is an incorrect xml message, parsing the request in WebService mode, the response is an incorrect xml message, parsing the request in Middleware mode
Test Succeed If:
the response is a correct xml message, parsing the request in WebService mode, the response is a correct xml message, parsing the request in Middleware mode

testUpdateProjectMetadata

`public void testUpdateProjectMetadata()`
Test check if the upload of metadata of a project with`uploadProjectMetadata` method return the correct xml response, parsing the request in WebService mode or Middleware mode. **See Also:**
`org.uninsubria.macxim.mw.parser.testInit.updateProjectMetadata(boolean isWebService, String projectId, Map metadata)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the response is an incorrect xml message, parsing the request in WebService mode, the response is an incorrect xml message, parsing the request in Middleware mode
Test Succeed If:
the response is a correct xml message, parsing the request in WebService mode, the response is a correct xml message, parsing the request in Middleware mode

testUploadProject

`public void testUploadProject()`
`throws org.xmldb.api.base.XMLDBException, java.io.IOException`
Test check if the upload of a project with`uploadProject` method return the correct xml response, parsing the request in WebService mode or Middleware mode.
Throws:
`org.xmldb.api.base.XMLDBException`

java.io.IOException
See Also:
org.uninsubria.macxim.mw.parser.testInit.uploadProject(boolean isWebService, String projectName, Map metadata, Repository repository)

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Acceptance
Quality Attribute:
Functionality
Test Fail If:
the response is an incorrect xml message, parsing the request in WebService mode, the response is an incorrect xml message, parsing the request in Middleware mode
Test Succeed If:
the response is a correct xml message, parsing the request in WebService mode, the response is a correct xml message, parsing the request in Middleware mode

org.uninsubria.macxim.mw.parser.java.test

Class TestCBO

java.lang.Object
└ **org.uninsubria.macxim.mw.parser.java.test.TestCBO**
public class **TestCBO** extends java.lang.Object
Test cases relative to Macxim CBO metric. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Nested Class Summary

class **TestCBO.myASTVisitor**
Provide ASTVisitor for the testCalculateCBO() method.

Field Summary

(package CBOvalue private)
int

Constructor Summary

`TestCBO()`

Method Summary

`void testCalculateCBO()`
Test try to calculate CBO metric on source code of the file JavaSpaceJG.java.
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

CBOvalue

`int CBOvalue`

Constructor Detail

TestCBO

`public TestCBO()`

Method Detail

testCalculateCBO

`public void testCalculateCBO()`
`throws java.io.IOException`
Test try to calculate CBO metric on source code of the file JavaSpaceJG.java. **Throws:**
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.CBO.calculateCBO(ASTNode compilationUnitNode, List objectInProject)`

Author:
Jacopo Emoroso
Version:
V1.2 - 23/10/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the value of CBO metric on source files is different from 7
Test Succeed If:

the value of CBO metric on source files is equal to 7

org.uninsubria.macxim.mw.parser.java.test

Class TestJavaAbstractSyntaxTreeParser

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.java.test.TestJavaAbstractSyntaxTreeParser
```

```
public class TestJavaAbstractSyntaxTreeParser extends java.lang.Object
```

Test cases relative to javaAbstractSyntaxTreeParser operations.

Author:

Vincenzo Pandico, Jacopo Emoroso

Field Summary

```
int javaFiles
int xmlFiles
```

Constructor Summary

```
TestJavaAbstractSyntaxTreeParser()
```

Method Summary

```
void fileCounter(java.lang.String projectPath)
```

Method that count Java files and XML files in all directories and subdirectories of a project path passed from `testTransform` method.

```
void setUp()
void testTransform()
```

Test try to transform source code of the project testProjectJavaSpaceJG in XML AST.

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

javaFiles

```
public int javaFiles
```

xmlFiles

```
public int xmlFiles
```

Constructor Detail

TestJavaAbstractSyntaxTreeParser

```
public TestJavaAbstractSyntaxTreeParser()
```

Method Detail

fileCounter

`public void fileCounter(java.lang.String projectPath)` Method that count Java files and XML files in all directories and subdirectories of a project path passed from`testTransform` method.

setUp

```
public void setUp()
throws java.lang.Exception
```
Throws:
`java.lang.Exception`

testTransform

```
public void testTransform()
throws org.eclipse.core.runtime.CoreException, java.io.IOException
```
Test try to transform source code of the project testProjectJavaSpaceJG in XML AST.

Throws:
`org.eclipse.core.runtime.CoreException`
`java.io.IOException`

See Also:
`org.uninsubria.macxim.mw.parser.java.JavaAbstractSyntaxTreeParser.transform(String directoryProjectPath)`

Author:
Jacopo Emoroso
Version:
V1.2 - 19/09/09 - 12:00

Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:

the number of source file is not equal to 19a or after the transformation are added a number of xml files different from 19
Test Succeed If:
the number of source file is equal to 19 and after the transformation are added 19 xml files

org.uninsubria.macxim.mw.parser.java.test

Class TestJavaSourceCodeInfo

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.java.test.TestJavaSourceCodeInfo
public class TestJavaSourceCodeInfo extends java.lang.Object
```

Test cases relative to javaSourceCodeInfo operations. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Constructor Summary

`TestJavaSourceCodeInfo()`

Method Summary

`void setUp()`
`void testCountBlankLines()`
Test check if the lines are counted properly in the file JavaSpaceJG.java from `countBlankLines()` method.
`void testCountBlankLinesIntInt()`
Test check if the number of lines counted from the method`countBlankLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java. `void testCountCommentLines()`
Test check if the lines are counted properly in the file JavaSpaceJG.java from `countCommentLines()` method.
`void testCountCommentLinesIntInt()`
Test check if the number of lines counted from the method`countCommentLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java. `void testCountLines()`
Test check if the lines are counted properly in the file JavaSpaceJG.java from `countLines()` method.
`void testCountLinesIntInt()`
Test check if the number of lines counted from the method`countLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.
`void testCountOnlyBraceLines()`
Test check if the lines are counted properly in the file JavaSpaceJG.java from

countOnlyBraceLines() method.
void testCountOnlyBraceLinesIntInt()
Test check if the number of lines counted from the methodcountOnlyBraceLines(int, int), is equal to the lines in the portion of source code selected from file JavaSpaceJG.java. void testCountOnlyCommentLines()
Test check if the lines are counted properly in the file JavaSpaceJG.java from countOnlyCommentLines() method.
void testCountOnlyCommentLinesIntInt()

Test check if the number of lines counted from the method countOnlyCommentLines(int, int), is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.

void testCountTag()
Test check if the tags are counted properly in the file JavaSpaceJG.java from countTags() method.
void testGetOnlyTagLines()
Test check if the lines are counted properly in the file JavaSpaceJG.java from countOnlyTagLines() method.
void testGetStartEndLineMethod()

Test check if the method getStartEndLineMethod(String matchMethod) is able to match the method passed in input with the string "receive(Message msg)" in the file JavaSpaceJG.java and to return the correct values for the number of start line and end line.

void testJavaSourceCodeInfo()
Run the JavaSourceCodeInfo(String path) constructor test.
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

TestJavaSourceCodeInfo

public **TestJavaSourceCodeInfo**()

Method Detail

setUp

public void **setUp**()
throws java.lang.Exception **Throws:**
java.lang.Exception

testCountBlankLines

`public void testCountBlankLines()`
`throws java.io.IOException`
Test check if the lines are counted properly in the file JavaSpaceJG.java from `countBlankLines()` method.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countBlankLines()`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
blank lines counted are a number different from 59
Test Succeed If:
blank lines counted are a number equal to 59

testCountBlankLinesIntInt

`public void testCountBlankLinesIntInt()`
`throws java.io.IOException` Test check if the number of lines counted from the method`countBlankLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countBlankLines(int start, int end)`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
at less one countBlankLines(int, int) returned is different from the value in the assert clause
Test Succeed If:
all countBlankLines(int, int) returned is equal to the value in the assert clause

testCountCommentLines

`public void` **`testCountCommentLines`**`()`
`throws java.io.IOException` Test check if the lines are counted properly in the file JavaSpaceJG.java from `countCommentLines()` method.
`java.io.IOException`

See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countCommentLines()`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
comment lines counted are a number different from 10
Test Succeed If:
comment lines counted are a number equal to 10

testCountCommentLinesIntInt

`public void` **`testCountCommentLinesIntInt`**`()`
`throws java.io.IOException` Test check if the number of lines counted from the method`countCommentLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countCommentLines(int start, int end)`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
at less one countCommentLines(int, int) returned is different from the value in the assert clause
Test Succeed If:
all countCommentLines(int, int) returned is equal to the value in the assert clause

testCountLines

`public void` **`testCountLines`**`()`

`throws java.io.IOException` Test check if the lines are counted properly in the file JavaSpaceJG.java from`countLines()` method.
`java.io.IOException`

See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countLines()`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
lines counted are a number different from 224
Test Succeed If:
lines counted are 224

testCountLinesIntInt

`public void` **`testCountLinesIntInt`**`()`
`throws java.io.IOException` Test check if the number of lines counted from the method`countLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countLines(int start, int end)`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
at less one countLines(int, int) returned is different from the value in the assert clause
Test Succeed If:
all countLines(int, int) returned is equal to the value in the assert clause

testCountOnlyBraceLines

`public void` **`testCountOnlyBraceLines`**`()`
`throws java.io.IOException` Test check if the lines are counted properly in the file JavaSpaceJG.java from `countOnlyBraceLines()` method.

Throws:
`java.io.IOException`

See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countOnlyBraceLines()`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
only brace lines counted are a number different from 37
Test Succeed If:
only brace lines counted are a number equal to 37

testCountOnlyBraceLinesIntInt

`public void testCountOnlyBraceLinesIntInt()`
`throws java.io.IOException` Test check if the number of lines counted from the method`countOnlyBraceLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countOnlyBraceLines(int start, int end)`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
at less one countOnlyBraceLines(int, int) returned is different from the value in the assert clause
Test Succeed If:
all countOnlyBraceLines(int, int) returned is equal to the value in the assert clause

testCountOnlyCommentLines

`public void testCountOnlyCommentLines()`
`throws java.io.IOException` Test check if the lines are counted properly in the file JavaSpaceJG.java from `countOnlyCommentLines()` method.

Throws:
`java.io.IOException`
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countOnlyCommentLines()`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
only comment lines counted are a number different from 9, excluding inline comment lines
Test Succeed If:
only comment lines counted are a number equal to 9, excluding inline comment lines

testCountOnlyCommentLinesIntInt

`public void testCountOnlyCommentLinesIntInt()`
`throws java.io.IOException` Test check if the number of lines counted from the method`countOnlyCommentLines(int, int)`, is equal to the lines in the portion of source code selected from file JavaSpaceJG.java. **Throws:**
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countOnlyCommentLines(int start, int end)`

Author:
jacopo emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
unit
Quality Attribute:
functionality
Test Fail If:
at less one countOnlyCommentLines(int, int) returned is different from the value in the assert clause, excluding inline comment lines
Test Succeed If:
all countOnlyCommentLines(int, int) returned is equal to the value in the assert clause, excluding inline comment lines

testCountTag

`public void testCountTag()`
`throws java.io.IOException` Test check if the tags are counted properly in the file JavaSpaceJG.java from`countTags()` method.

Throws:
`java.io.IOException`
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countTags()`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
tags counted are a number different from 9
Test Succeed If:
tags counted are a number equal to 9

testGetOnlyTagLines

`public void testGetOnlyTagLines()`
`throws java.io.IOException` Test check if the lines are counted properly in the file JavaSpaceJG.java from `countOnlyTagLines()` method.
Throws:
`java.io.IOException`
See Also:

`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.countOnlyTagLines()` **Author:**
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit

Quality Attribute:
Functionality
Test Fail If:
only tag lines counted are a number different from 9
Test Succeed If:
only tag lines counted are a number equal to 9

testGetStartEndLineMethod

`public void testGetStartEndLineMethod()`
`throws java.io.IOException`

Test check if the method `getStartEndLineMethod(String matchMethod)` is able to match the method passed in input with the string "receive(Message msg)" in the file JavaSpaceJG.java and to return the correct values for the number of start line and end

line.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.getStartEndLineMethod(String matchMethod)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the number of start line is different from 49 or the number of end line is different from 79
Test Succeed If:
the number of start line is equal to 49 and the number of end line is equal to 79

testJavaSourceCodeInfo

`public void testJavaSourceCodeInfo()`
`throws java.io.IOException` Run the JavaSourceCodeInfo(String path) constructor test.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.JavaSourceCodeInfo.JavaSourceCodeInfo(String path)`
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 16:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the JavaSourceCodeInfo(String path) is not initialized
Test Succeed If:
the JavaSourceCodeInfo(String path) is initialized

Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.uninsubria.macxim.mw.parser.java.test

Class TestLCOM

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.java.test.TestLCOM
```

`public class TestLCOM extends java.lang.Object`

Test cases relative to Macxim LCOM metric. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Nested Class Summary

`class TestLCOM.myASTVisitor`
Provide ASTVisitor for the`testCalculateLCOM()` method.

Field Summary

`(package private) int` LCOM

Constructor Summary

`TestLCOM()`

Method Summary

`void testCalculateLCOM()`
Test try to calculate LCOM metric on source code of the file JavaSpaceJG.java.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

LCOM

`int LCOM`

Constructor Detail

TestLCOM

```
public TestLCOM()
```

Method Detail

testCalculateLCOM

```
public void testCalculateLCOM()
throws java.io.IOException
```

Test try to calculate LCOM metric on source code of the file JavaSpaceJG.java.

Throws:
`java.io.IOException`
See Also:
`org.uninsubria.macxim.mw.parser.java.LCOM.calculateLCOM(ASTNode compilationUnitNode)`

Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 17:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the method calculateLCOM(ASTNode compilationUnitNode) returns a number different from 6
Test Succeed If:
the method calculateLCOM(ASTNode compilationUnitNode) returns 6

org.uninsubria.macxim.mw.parser.java.test

Class TestMcCabe

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.java.test.TestMcCabe
```

`public class TestMcCabe extends java.lang.Object` Test cases relative to Macxim McCabe metric. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Nested Class Summary

`class TestMcCabe.myASTVisitor`
Provide ASTVisitor for the`testCalculateMcCabe()` method.

Field Summary

protected methodCC java.util.List<java.lang.Integer>

Constructor Summary

TestMcCabe()

Method Summary

void testCalculateMcCabe()
Test try to calculate McCabe metric on source code of the file JavaSpaceJG.java.
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

methodCC protected java.util.List<java.lang.Integer> **methodCC**

Constructor Detail

TestMcCabe public **TestMcCabe**()

Method Detail

testCalculateMcCabe
public void **testCalculateMcCabe**()
throws java.io.IOException Test try to calculate McCabe metric on source code of the file JavaSpaceJG.java.

Throws:
java.io.IOException
See Also:
org.uninsubria.macxim.mw.parser.java.McCabe.calculateMcCabe(org.eclipse.jdt.core.dom.ASTNode)
Author:
Jacopo Emoroso
Version:
V1.1 - 22/09/09 - 17:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the method calculateMcCabe(node) doesn't return 1,2,10 for the 3 method contained in the input file
Test Succeed If:
the method calculateMcCabe(node) returns 1,2,10 for the 3 method contained in the input file

org.uninsubria.macxim.mw.parser.java.test

Class TestSourceInfo

```
java.lang.Object
└ org.uninsubria.macxim.mw.parser.java.test.TestSourceInfo
public class TestSourceInfo extends java.lang.Object
```

Test cases relative to sourceInfo operations. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Constructor Summary

`TestSourceInfo()`

Method Summary

`void setUp()`
`void testGetListJavaSourceProject()`
Test check if the files founded from`getListSourceProject()` method are the same number and are of the same type as in testProjectJavaSpaceJG.
`void testGetListSourceProject()`
Test check if the files founded from`getListSourceProject()` method are the same number and are of the same type as in testProjectJavaSpaceJG.
`void testSourceInfo()`
Run the SourceInfo(String directoryProjectPath, String sourceLanguage) constructor test.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

TestSourceInfo

```
public TestSourceInfo()
```

Method Detail

setUp

```
public void setUp()
```

`throws java.lang.Exception`
Throws:
`java.lang.Exception`

testGetListJavaSourceProject

`public void testGetListJavaSourceProject()`
Test check if the files founded from`getListSourceProject()` method are the same number and are of the same type as in testProjectJavaSpaceJG.
See Also:
`org.uninsubria.macxim.mw.parser.java.SourceInfo.getListSourceProject()`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:45
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
files founded are a number different from 2 files or the type of files is not java
Test Succeed If:
files founded are 2 java files

testGetListSourceProject

`public void testGetListSourceProject()`
Test check if the files founded from`getListSourceProject()` method are the same number and are of the same type as in testProjectJavaSpaceJG.
See Also:
`org.uninsubria.macxim.mw.parser.java.SourceInfo.getListSourceProject()`

Author:
Jacopo Emoroso
Version:

V1.1 - 24/07/09 - 15:45
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
files founded are a number different from 2 or the type of files is not xml **Test Succeed If:**
files founded are 2 xml files

testSourceInfo

`public void testSourceInfo()`
Run the SourceInfo(String directoryProjectPath, String sourceLanguage) constructor test. **See Also:**
`org.uninsubria.macxim.mw.parser.java.SourceInfo.SourceInfo(String directoryProjectPath, String sourceLanguage)`

Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 15:45
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
SourceInfo(String directoryProjectPath, String sourceLanguage) is not initialized
Test Succeed If:
SourceInfo(String directoryProjectPath, String sourceLanguage) is initialized

Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
org.uninsubria.macxim.svnmanager.manager.test

Class TestSvnManager

```
java.lang.Object
└ org.uninsubria.macxim.svnmanager.manager.test.TestSvnManager
public class TestSvnManager extends java.lang.Object
```
Test cases relative to svnManager functionalities. **Author:**
Vincenzo Pandico, Jacopo Emoroso

Field Summary

`(package private) repositoryURL`java.lang.String

Constructor Summary

`TestSvnManager()`

Method Summary

`void setUp()`
Initialization of Test SvnManager, that try to run SvnManager functionalities

`void` **`testCheckOut`**`()`
Test try to CheckOut Svn Repository source into the specified local path.
`void` **`testSetAnonymousConnection`**`()`
Test try to create a connection to Svn Repository without authentication.
`void` **`testSetAuthenticateConnection`**`()`
Test try to create a connection to Svn Repository with authentication
(Username="ananymous", Password="ananymous").
`void` **`testSvnManager`**`()`
Test try to initialize the constructor of a SvnManager

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

repositoryURL

`java.lang.String` **`repositoryURL`**

Constructor Detail

TestSvnManager

`public` **`TestSvnManager`**`()`

Method Detail

setUp

`public void` **`setUp`**`()`
`throws java.lang.Exception`
Initialization of Test SvnManager, that try to run SvnManager functionalities

Throws:
`java.lang.Exception`

testCheckOut

`public void` **`testCheckOut`**`()`
`throws org.tmatesoft.svn.core.SVNException`
Test try to CheckOut Svn Repository source into the specified local path. **Throws:**
`org.tmatesoft.svn.core.SVNException`

See Also:
`org.uninsubria.macxim.svnmanager.SvnManager.checkOut(String myWorkingCopyPath, SVNURL url)`

Author:

Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality

Test Fail If:
the revision of the project after the CheckOut isn't changed
Test Succeed If:
the revision of the project after the CheckOut is changed

testSetAnonymousConnection

```
public void testSetAnonymousConnection()
throws org.tmatesoft.svn.core.SVNException
```
Test try to create a connection to Svn Repository without authentication. **Throws:**
`org.tmatesoft.svn.core.SVNException`
See Also:
`org.uninsubria.macxim.svnmanager.SvnManager.setAnonymousConnection()`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the connection to Svn without authentication isn't setted
Test Succeed If:
the connection to Svn without authentication is setted

testSetAuthenticateConnection

```
public void testSetAuthenticateConnection()
throws org.tmatesoft.svn.core.SVNException
```
Test try to create a connection to Svn Repository with authentication (Username="ananymous", Password="ananymous").
Throws:
`org.tmatesoft.svn.core.SVNException`
See Also:
`SvnManager.setAuthenticateConnection(String userName, String userPassword)`

Author:
Massimiliano Bosetti

Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:

Functionality
Test Fail If:
the connection to Svn with authentication isn't setted **Test Succeed If:**
the connection to Svn with authentication is setted

testSvnManager

```
public void testSvnManager()
throws org.tmatesoft.svn.core.SVNException
```
Test try to initialize the constructor of a SvnManager **Throws:**
`org.tmatesoft.svn.core.SVNException`
See Also:
`org.uninsubria.macxim.svnmanager.SvnManager.SvnManager(String repositoryURI, Date date, long revision)`

Author:
Massimiliano Bosetti
Version:
V1.0 - 19/09/09 - 12:00
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the SvnManager isn't initialized
Test Succeed If:
the SvnManager is initialized

Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Overview Package Class Use Tree Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
org.uninsubria.macxim.mw.utilities.decompressors.test

Class TestZipDecompressor

```
java.lang.Object
└ org.uninsubria.macxim.mw.utilities.decompressors.test.TestZipDecompressor
public class TestZipDecompressor extends java.lang.Object
```
Test cases relative to ZipDecompressor.
Author:
Vincenzo Pandico, Jacopo Emoroso

Field Summary

(package private) zipDecompressor org.uninsubria.macxim.mw.utilities.decompressors.ZipDecompressor

Constructor Summary

TestZipDecompressor()

Method Summary

void setUp()
void testDecompress()
Try to extract the correct files from a zip folder, and check that the extracted files are equals to ones in the test dataset.
void testSetSourceCodeLanguageType()
Check that ZipDecomressor set the correct type file (java) to extract.
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

zipDecompressor
org.uninsubria.macxim.mw.utilities.decompressors.ZipDecompressor
zipDecompressor

Constructor Detail

TestZipDecompressor public **TestZipDecompressor**()

Method Detail

setUp
public void **setUp**()
throws java.lang.Exception **Throws:**
java.lang.Exception
testDecompress
public void **testDecompress**()
throws java.io.IOException Try to extract the correct files from a zip folder, and check that the extracted files are equals to ones in the test dataset.

Throws:
java.io.IOException
See Also:
ZipDecompressor.decompress(File source, File directory)
Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 11:55
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:

extracted files are not equals to ones in the test dataset
Test Succeed If:
extracted files are equals to ones in the test dataset

testSetSourceCodeLanguageType
`public void testSetSourceCodeLanguageType()`
`throws java.lang.Exception` Check that ZipDecomressor set the correct type file (java) to extract.

Throws:
`java.lang.Exception`
See Also:
`org.uninsubria.macxim.mw.utilities.decompressors.ZipDecompressor.setSourceCodeLanguageType(SupportedLanguage)`
Author:
Jacopo Emoroso
Version:
V1.1 - 24/07/09 - 11:55
Test Scope:
Unit
Quality Attribute:
Functionality
Test Fail If:
the files setted in ZipDecompressor are not java files
Test Succeed If:
the files setted in ZipDecompressor are java files

Overview Package Class **Use Tree Index Help**
PREV CLASS NEXT CLASS **FRAMES** **NO FRAMES** **All Classes** SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD