# T-DOC: a Tool for the Automatic Generation of Testing Documentation for OSS Products

Sandro Morasca, Davide Taibi, and Davide Tosi

Università degli Studi dell'Insubria,
Dipartimento di Informatica e Comunicazione, Via Mazzini, 21100 Varese, Italy
{sandro.morasca,davide.taibi,davide.tosi}@uninsubria.it

**Abstract.** In the context of Open Source Software (OSS), the lack of project documentation is one of the most challenging problems that slows down the widespread diffusion of OSS products. The difficulty of providing up-to-date and reasonable documentation for OSS products relates to two main reasons. First, documenting development activities and technological issues is viewed as a tedious and unrewarding task. Second, data and information about an OSS project (such as source code, project plans, testing requirements, etc.) are scattered and shared via unstructured channels such as unofficial forums and mailing lists.
In this paper, we focus on technical documentation related to testing activities. In this context, the lack of documentation is exacerbated due to the use of the available testing methods that drastically increase code fragmentation. We propose T-doc, a tool that simplifies the generation of testing documentation. In particular, T-doc supports (1) the automatic generation of test cases documentation, (2) the generation of reports about test case results, and (3) the archiving of testing documents in central repositories. The automatic generation of documentation is facilitated by the adoption of built-in testing methods that simplify the aggregation of testing data.
We apply the tool to the OSS RealEstate Java application to show the applicability and the real benefits of our solution.

**Key words:** Open Source Software testing, testing documentation, testing tools

## 1 Introduction

Open Source Software (OSS) is experiencing an increasing diffusion and popularity in industrial sectors. However, this spreading is slowed down by the frustration a lot of potential users have when they start evaluating an OSS product that they would like to adopt. This is primarily due to the lack of reasonable and up-to-date user documentation that deeply describes the intent and the technical aspects of the project.

Most of the available OSS projects are currently released without up-to-date user manuals and technical documents. The lack of documentation in OSS is even more serious in the context of testing activities. It is very rare to find

well-structured documents, manuals, and reports about all the testing phases performed during the development of OSS products. Documenting OSS projects is a tedious and unrewarding task that is made more complicated by the scattering of data and information typical of OSS projects.

In this paper, we focus on the problem of documenting testing activities and we propose a tool (we 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 we 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 [3]. T-doc provides a three-layered support:

- automatic generation of test cases documentation (in a java-doc like style);
- automatic generation of suggestions about integration and regression testing activities that should be performed by each developer and for each component of the project;
- automatic generation of reports about the results of test suites 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. This paper is a step towards our final goal, which is the development of a standard framework that OSS developers can use whenever they start testing their OSS products. In this paper, we apply an initial implementation of T-doc to the RealEstate Java application [2] to show the simplicity, the real benefits, and the level of automation provided by our solution.

The paper is structured as follows: Section 2 reports the analysis we conducted to confirm the low availability of testing documentation, and discusses the limits of a set of existing testing tools; Section 3 introduces the motivations that are at the basis for adopting built-in testing in the context of OSS products; Section 4 separately discusses the three layers of the T-doc tool, and shows how T-doc comes into play when applied to the RealEstate Java application; and finally we conclude in Section 5.

## 2 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[1]. The first analysis aimed to identify the importance the factor "avail-

[1] an extensive report of these experiences can be found in www.qualipso.eu/node/45 and /node/84

ability of technical documentation / user manual" has 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 when 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 primary 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. Finally, the lack of tools, which support and automate the documentation of testing activities, leaves too much effort to the side of developers. The results obtained by our second exploration are in contrast with the requirements OSS users have. This analysis confirms our intuition and demonstrates the need for a tool that supports the automatic generation of testing documentation.

Currently, open source tools or frameworks that support the whole documentation of testing activities are not yet fully available. The famous portal [www.opensourcetesting.org] gathers a lot of testing tools that support a specific aspect of the test life cycle, but none of them are able to manage and create the documentation, the results report and the collection of these information. For example, Testopia [www.mozilla.org/projects/testopia] is a test case management extension for Bugzilla that tracks test cases and allows for testing organizations to integrate bug reporting with their test case run results. However, Testopia covers only a part of the functionalities provided by T-doc. Fitness [http://fitnesse.org] is a software development collaboration tool, which simplifies the management of testing documentation, test reports and the collaborative definition of acceptance tests. T-doc, is able to automatically generate testing documentation and it is not limited to acceptance tests. Moreover, T-doc is able to automatically suggest the integration and regression testing activities that should be performed. Other tools, such as TPTP [www.eclipse.org/tptp/] or Salome-TMF [https://wiki.objectweb.org/salome-tmf/], are complex frameworks that cover the entire test life cycle but are not able to automatically create testing documentation.

The next section discusses why a built-in testing method is preferred to classic testing solutions.

```
1  Class class_name {
2    // application interface
3    Data declaration;
4    Constructor declaration;
5    Destructor declaration;
6    Methods declaration;
7
8    // testing interface
9    TestCases declaration;
10
11   // application code
12   Constructor;
13   Destructor;
14   Methods;
15
16   // testing code
17   TestCases;
18 }
```

**Listing 1.1.** Code excerpt of a BIT component

## 3 Built-in Test in OSS

Built-in self-test (BIST) and Built-in test (BIT) approaches for software systems originated in the context of component-based systems to simplify the integration of third-party black-box components and enhance software maintainability [9]. A BIT component (or BIT class) is a traditional component that puts together applicative code with testing code [3]. A BIT component can operate in a normal mode (i.e., testing capabilities are switched off 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 application or the testing interface, respectively. Listing 1.1 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.

In the context of OSS, the heterogeneity of the developers/contributors increases the fragmentation of the source code and makes unfeasible the adoption of available testing methods, programming rules, and testing tools that could favor the whole comprehension of fragmented testing activities. Simple programming rules (as shown in Listing 1.1) may help standardize a common programming style that can improve the testing activity, decrease the testing effort, and simplify the generation of testing documentation. Whenever a developer/contributor of an OSS product introduces or modifies a functionality of a component, he or 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 1).

Putting together application code and testing code into single classes has several advantages: (1) it improves the visibility and inheritance of test cases. Test cases are coded as classic methods thus, when a class extends another class, the

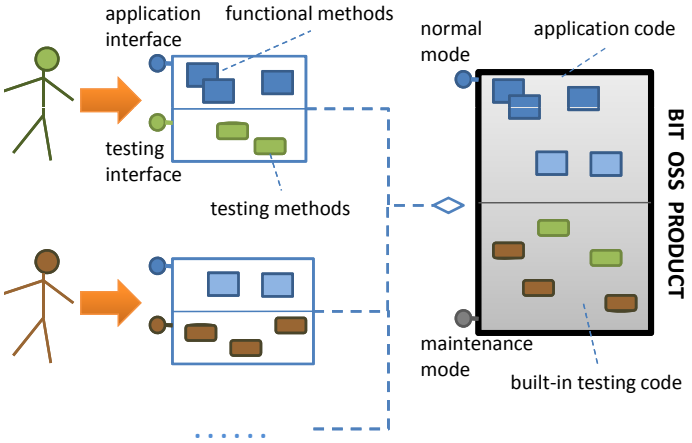


**Fig. 1.** Aggregating components into an OSS product with BIT abilities

former inherits not only the application methods but also the test case methods. This simplifies the reuse of available test cases; (2) it favors the standardization of testing interfaces. Test cases are developed following the coding rules of the target programming language in use for the application, thus limiting the creativity of the developers. This improves the readability of the testing code; (3) it increases the aggregation of data. Test cases are grouped into single classes instead of into different packages, components, or libraries. This simplifies the discovery of testing data and their correlation with coding elements; (4) moreover, the documentation of test activities and the report of test case results is made easier, thus simplifying regression testing activities. Regression testing is made upon the availability of test cases and test results. The more test cases and test results are not available or they are disaggregated, the more the regression testing activity is tricky; (5) it favors run-time testing [8]: the system can be executed at run-time in maintenance modality [7], thus simplifying the detection of bugs that are undetectable in a controlled testing environment. In OSS, often components are separately tested at development time by each contributor that develops a small unit and tests its behavior in isolation. This leaves undetected a lot of integration failures. Moreover with BIT, the test suite can be executed over different hw/sw platform configurations, thus simplifying system, configuration and performance testing. Every time a user installs the application on his environment, he/she becomes a new tester of the application and he/she uses his/her hw/sw configuration as a new scaffolding of the testing activity. 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 tool: run-time testing can move the system in an in-

consistent 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 aware that the OSS product is under BIT, so they can block the BIT abilities if they so wish, 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 classic testing mechanisms, is a valid way to support and simplify the generation and the gathering of testing documentation in the domain of OSS.

## 4 The T-doc Tool

Here, we present the architecture of the T-doc tool 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. Figure 2 shows a high level architecture of T-doc.

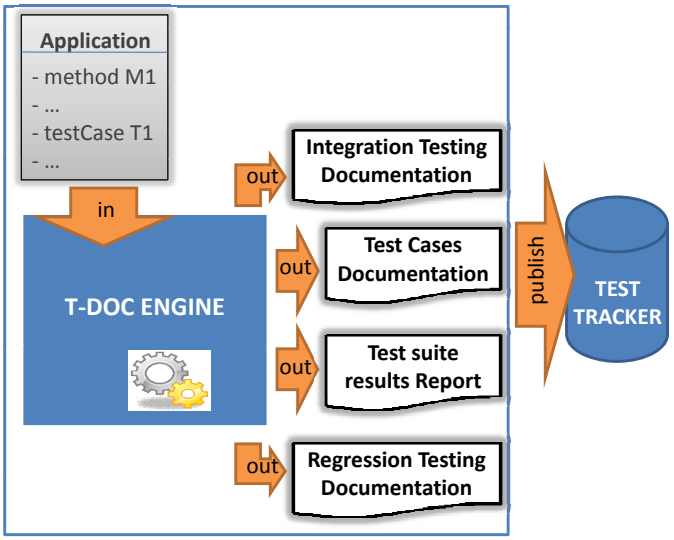


**Fig. 2.** High level architecture of the T-doc framework

### 4.1 Test Case Documentation

This first layer of support aims at simplifying and automating test case and test suite documentation generation. The generated documentation should increase the readability of the technical aspects of each test case, and should favor an overall comprehension of the testing activity. To allow for the automation 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 behavior) and keywords in a way similar to the 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 processed by the T-doc engine to generate the test case documentation much the same way as 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`).

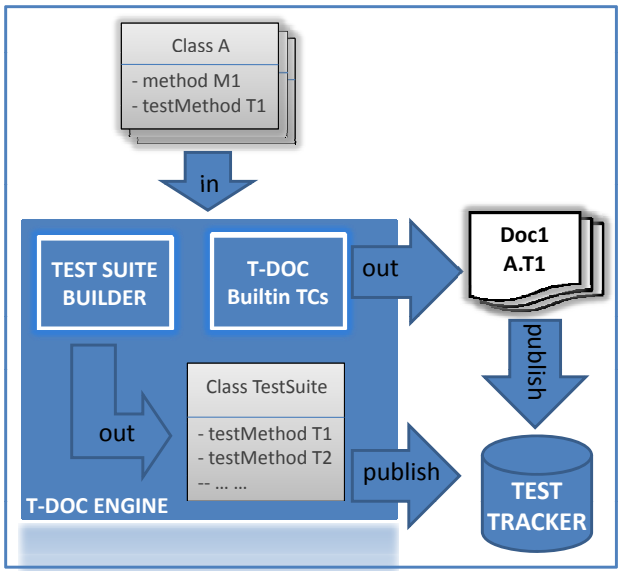


**Fig. 3.** Architecture of the first T-doc layer

To minimize the effort of developers and contributors in writing testing documentation, favor standardization, and avoid subjective interpretations of data, we clearly define a set of new conventions and a set of ad-hoc 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 4.

In compliance with Javadoc, the conventions we defined 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 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 tags, useful for commenting a test case, are listed below:

```
@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 aspect addressed: performance, etc.)
@scope  (specify the test case purpose: unit, integration, etc.)
@author   (author name/surname)
@version  (version number + checkout date)
@see  package.Class#method(Type,...) (ref to the function under test)
```

Figure 3 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`.

To favor 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 [2]. The RealEstate is a Java application created at North Carolina State University that reproduces the Monopoly game. The RealEstate application will be used throughout the whole paper as proof-of-concept of our work. Figure 4 shows the source code of the built-in test case surrounded by a T-doc comment and T-doc tags. The purpose of this Figure is not to present the internal code of the test, but to highlight the structure of a T-doc comment.

```
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 $
 *   @scope          unit testing
 *   @author         Davide Tosi
 *   @version        1.0.2  06/02/09
 *   @see            edu.ncsu.realestate.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());
}
```

**Fig. 4.** A built-in test case with T-doc comments for the RealEstate application

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

```
ID001:: UNIT Test: testGainMoneyCardAction
V1.0.2 06-02-09

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.

Succeeds if: getMoney() returns a value=1550$
Fails if: getMoney() returns a value!=1550$
See: edu.ncsu.realestate.MoneyCard()
```

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 and also the readability of the documentation. Currently, this T-doc module has been fully implemented and its is fully compatible with the Eclipse IDE.

### 4.2 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 dependencies among methods and components must be detected by the T-doc engine and visually reported to the developer. The T-doc engine uses the idea of *change points* and *call graphs* [4] [5] 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 the starting point to create the suggestions for integration and regression testing activities.

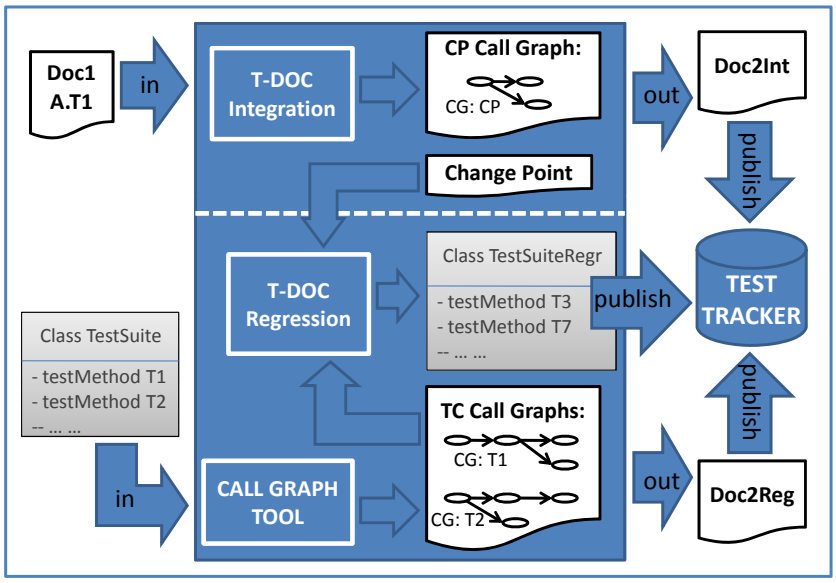


**Fig. 5.** Architecture of the second T-doc layer

Figure 5 shows the three main modules of this layer: the `T-doc Integration` module, the `T-doc Regression` module and the `Call Graph` tool.

The `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 uses class B if objects of class A make method calls on objects of class B, or if objects of A contain references to objects of B [6]. The `T-doc Integration` takes as input the documentation generated by the `T-doc TCs` module (`Doc1 A.T1`) and automatically generates the call graph for the change point (`CP`) that is related to the documented test case. To avoid graph size explosion, we chose to limit the computation to the third level of method's dependencies. We are conducting additional experiments to understand the code coverage we obtain with this limit.

Referring to our RealEstate example of Figure 4, the OSS contributor is working on the `MoneyCard` class, by modifying the `applyAction()` method

and 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 6 shows the result of this computation (`Doc2Int`). The root of the graph is the CP `applyAction()`, while leaves are the methods that directly or indirectly interact with the `applyAction()` method. The `T-doc Integration` module integrates the functionalities provided by `CallGraph` [www.certiv.net/projects/] to automatically create call graphs starting from a change point.

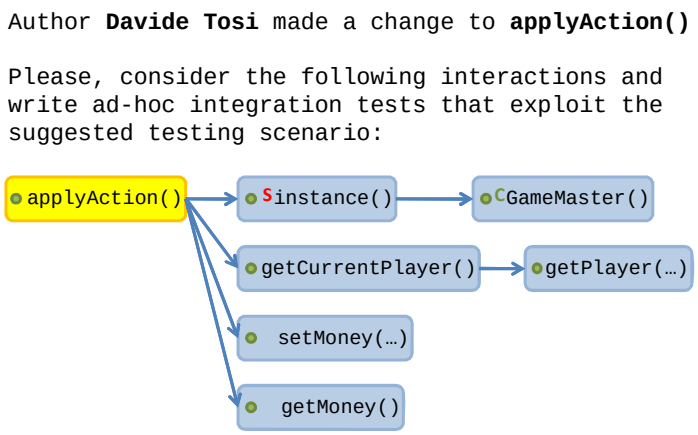


**Fig. 6.** Generated integration testing scenario for the `testGainMoneyCardAction()`

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 [`sourceforge.net`] require 45 mins in a fully dedicated machine. Moreover, other problems are: who runs the test suite? Where to store and collect the test cases that should be re-executed? When must the test cases be rerun? Where are the results of the test suite execution reported? 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. Here, we show the algorithm that the `T-doc Regression` module uses to detect the subset of meaningful test cases:

```
Input: test cases, CP
Output: documentation of the subset of meaningful regression tests

1. derive the call graph for each test case ending at the 3rd level
   of dependencies;
2. select a graph as starting entry;
3. scan the graph 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 as input, from the `Call Graph Tool`, 30 graphs and generated the following documentation (`Doc2Reg`). For space reason, we do not show the complete set of graphs computed by the T-doc engine.

```
Doc2Reg:
 This is the set of regression test cases
 for the applyAction() change point:
 01) testGainMoneyCardAction()
 02) testMovePlayerCardAction()
 03) testLoseMoneyCardAction()
 04) testJailCardAction()
 05) testJailCardUI()
 06) testLoseMoneyCardUI()
 07) testMovePlayerCardUI()
```

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 Test Case Execution Report

This third layer of support aims at homogenizing and collecting all the outputs coming from the T-doc tool and the results obtained by the execution of the test cases. In this section, we only introduce the design of this layer since its 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 for 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 behavior 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 identical to the tags we defined in Section 4.1. 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 according to the purpose of test cases).

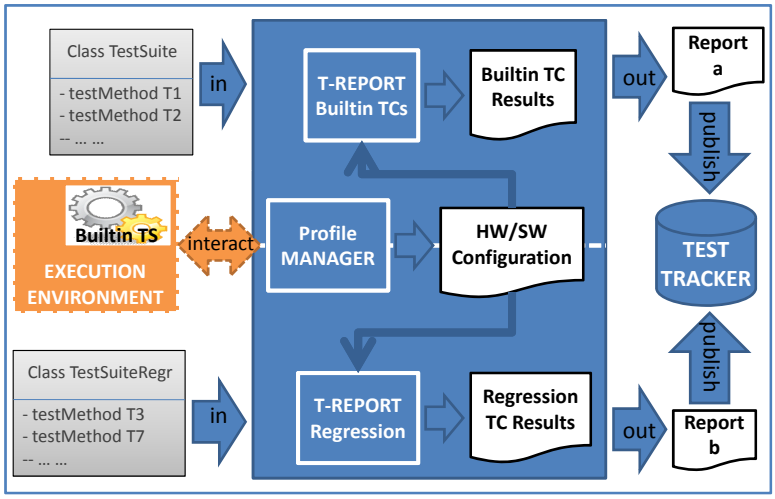


**Fig. 7.** Architecture of the third T-doc layer

As mentioned in Section 3, built-in test cases favor the execution of run-time testing [7]. 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 7 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.

### 4.4 Validation Remarks

Though the RealEstate demo application has shown the feasibility and the benefits of the T-doc tool, we are extending the validation of the tool with additional case studies. For example, in our labs, we are implementing a complex OSS project to validate the approach and to understand its potentialities. The project (we called MacXim) is a static-analysis tool (15000 LOC in 118 classes) that exploits the solution presented in this paper [1]. The MacXim test suite (composed of acceptance, unit, integration and regression tests) has been designed with in mind the guidelines proposed in this paper, and each test case has been documented with a T-doc comment that describes the purpose of the test.

This controlled project will provide important feedbacks about the potentialities and the weaknesses of the T-doc tool, and will be the basis for developing a stable tool that will be fully exploited in real-life OSS projects and in uncontrolled development environments.

## 5 Conclusions and Future Work

In this paper, we proposed T-doc, a tool that simplifies the generation of testing documentation in the context of OSS projects. We showed how T-doc 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. The automatic generation of documentation is facilitated by the adoption of built-in testing methodologies that simplify the aggregation of testing data. To understand the T-doc working in practice, we applied the tool to the OSS RealEstate Java application.

Currently, we are integrating all the modules of the T-doc tool and we are validating T-doc with the real-life Java application MacXim.

## 6 Acknowledgments

The research presented in this paper has been partially funded by the IST project QualiPSo (http://www.qualipso.eu/), sponsored by the EU in the 6th FP (IST-034763); the FIRB project ARTDECO, sponsored by the Italian Ministry of Education and University; and the projects "Elementi metodologici per la descrizione e lo sviluppo di sistemi software basati su modelli" and "La qualità nello sviluppo software," funded by the Università degli Studi dellInsubria.

## References

1. MacXim: a static code analysis tool. Web published: blinded. Accessed: December 2009.
2. The RealEstate demo application. Web published: http://agile.csc.ncsu.edu/SEMaterials/realestate/. Accessed: December 2009.

3. S. Beydeda. Research in testing COTS components - built-in testing approaches. In *Proceedings of the ACM/IEEE International Conference on Computer Systems and Applications (AICCSA)*, pages 101–104, 2005.
4. C. Mao, Y. Lu, and J. Zhang. Regression testing for component-based software via built-in test design. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1416–1421, 2007.
5. A. Orso, 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. In *Proceedings of the IEEE International Conference on Software Manintenance (ICSM)*, pages 716–725, 2001.
6. M. Pezzè and M. Young. *Software Testing And Analysis. Process, Principles, and Techniques*. Wiley, 2007.
7. D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka. The MORABIT approach to runtime component testing. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pages 171–176, 2006.
8. J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component-based software systems. *Software Quality Control*, 10(2):115–133, 2002.
9. Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, pages 186–192, 1999.