Model-based Testing: Next Generation Functional Software Testing

By Dr. Bruno Legeard

Model-based testing (MBT) is an increasingly widely-used technique for automating the generation and execution of tests. There are several reasons for the growing interest in using model-based testing:

- The complexity of software applications continues to increase, and the user's aversion to software defects is greater than ever, so our functional testing has to become more and more effective at detecting bugs;
- The cost and time of testing is already a major proportion of many projects (sometimes exceeding the costs of development), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically as well as executing them automatically.
- The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good ROI;
-

Model-based testing renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation.

This paper addresses these points by giving a realistic overview of model-based testing and its expected benefits. We discuss ***what*** model-based testing is, ***how*** you have to organize your process and your team to use MBT and detail a complete example from business requirements to automated test repository using a fully supported MBT process.

What is MBT?

Model-based testing refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases.

Therefore, the key points of model-based testing are the modeling principles for test generation, the test generation strategies and techniques, and the concretization of abstract tests into concrete, executable tests. A typical deployment of MBT in industry goes through the four stages shown in Figure 1:

Dagstuhl Seminar Proceedings 10111
Practical Software Testing : Tool Automation and Human Factors
http://drops.dagstuhl.de/opus/volltexte/2010/2620

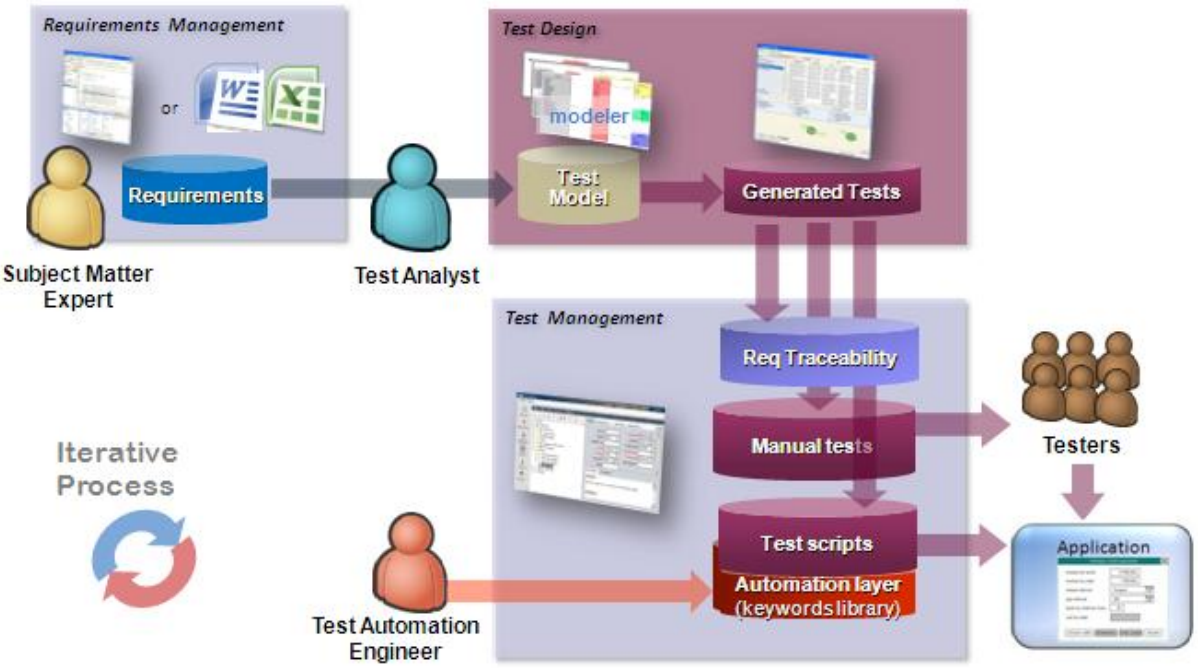

Figure 1. The MBT process.

1. **Design a Test Model.** The model, generally called the *test model*, represents the expected behavior of the system under test (SUT). Standard modeling languages such as UML are used to formalize the control points and observation points of the system, the expected dynamic behavior of the system, the business entities associated with the test, and some data for the initial test configuration. Model elements such as transitions or decisions are linked to the requirements, in order to ensure bi-directional traceability between the requirements and the model, and later to the generated test cases. Models must be precise and complete enough to allow automated derivation of tests from these models;
2. **Select some Test Generation Criteria.** There are usually an infinite number of possible tests that could be generated from a model, so the test analyst chooses some Test Generation Criteria to select the highest priority tests, or to ensure good coverage of the system behaviors. One common kind of test generation criteria is based on structural model coverage, using well known test design strategies such as equivalence partitioning, cause-effect testing, pair-wise testing, process cycle coverage, or boundary value analysis (see [1] for more details on these strategies). Another useful kind of test generation criteria ensures that the generated test cases cover all the requirements, perhaps with more tests generated for requirements that have a higher level of risk. In this way, model-based testing can be used to implement a requirement and risk-based testing approach. For example, for a non-critical application, the test analyst may choose to generate just one test for each of the nominal behaviors in the model and each of the main error cases; but for one of the more critical requirements, she/he could apply more demanding coverage criteria such as all loop-free paths, to ensure that the businesses processes associated with that part of the test model are more thoroughly tested;
3. **Generate the tests.** This is a fully automated process that generates the required number of (abstract) test cases from the test model. Each generated abstract test case is typically a sequence of high-level SUT actions, with input parameters and expected output values for

each action. These generated test sequences are similar to the high-level test sequences that would be designed manually in action-word testing [2]. They are easily understand by humans and are complete enough to be directly executed on the SUT by a manual tester. The test model allows computing the expected results and the input parameters. Data table may be use to link some abstract value from the model with some concrete test value. To make them executable using a test automation tool, a further concretization phase automatically translates each abstract test case into a concrete (executable) scripts [3], using a user-defined mapping from abstract data values to concrete SUT values, and a mapping from abstract operations into SUT GUI actions or API calls. For example, if the test execution is via the GUI of the SUT, then the action words are linked to the graphical object map, using a test robot such as HP QuickTest **Professional**, IBM Rational Functional Tester or the open-source robot Selenium. If the test execution of the SUT is API-based, then the action words need to be implemented on this API. This can be a direct mapping or a more complex automation layer. The expected results part of each abstract test case is translated into oracle code that will check the SUT outputs and decide on a test pass/fail verdict. The tests generated from the test model may be structured into multiple test suites, and published into standard test management tools such as HPQuality Center, IBM Rational Quality Manager or the open-source tool TestLink. Maintenance of the test repository is done by updating the test model, then automatically regenerating and republishing the test suites into the test management tools;

4. **Execute the Tests.** The generated concrete tests are typically executed either manually or within a standard automated test execution environment, such as HP QuickTest Professional or IBM Rational Functional Tester. Either way, the result is that the tests are executed on the SUT, and we find that some tests pass and some tests fail. The failing tests indicate a discrepancy between the SUT and the expected results designed in the test model, which then needs to be investigated to decide whether the failure is caused by a bug in the SUT, or by an error in the model and/or the requirements. Experience shows that model-based testing is good at finding SUT errors, but is also highly effective at exposing requirements errors [1] even far before executing a single test (thanks to the modeling phase).

Requirements traceability

The automation of bidirectional traceability between requirements and test cases is a key aspect of the added-value of MBT. *Bidirectional traceability* is the ability to trace links between two parts of the software development process with respect to each other. The starting point of the MBT process is, as usual, the informal functional requirements, use cases, descriptions of business processes and all other factors that provide the functional description of the application being tested. To be effective, requirements traceability implies that the requirements repository should be structured enough so that each individual requirement can be uniquely identified. It is desirable to link these informal requirements to the generated tests, and to link each generated test to the requirements that it tests.

A best practice in MBT, supported by most of the tools on the market, consists in linking model elements such as decision points and transitions to the relevant requirements. From these links in the test model, test generation tools ensure the automatic generation and maintenance of the traceability matrix between requirements and test cases.

Test repository and test management tools

The purpose of generating tests from the test model is to produce the test repository. This test repository is typically managed by a test management tool, such as HP Quality Center, IBM Rational Quality Manager or the open-source tool TestLink. The goal of such a tool is to help organize and execute test suites (groups of test cases), both for manual or automated tests.

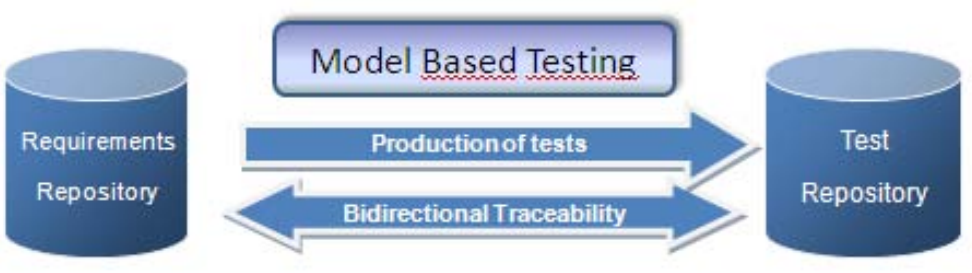

Figure 2. Relationship between both repositories (tests and requirements).

In the MBT process, the test repository documentation is fully managed by automated generation(from the test model): documentation of the test design steps, requirements traceability links, test scripts and associated documentation are automatically provided for each test case. Therefore, the maintenance of the test repository needs to be done in the test model.

Roles in the MBT process

The MBT process involves three main kinds of roles (see Figure 3).

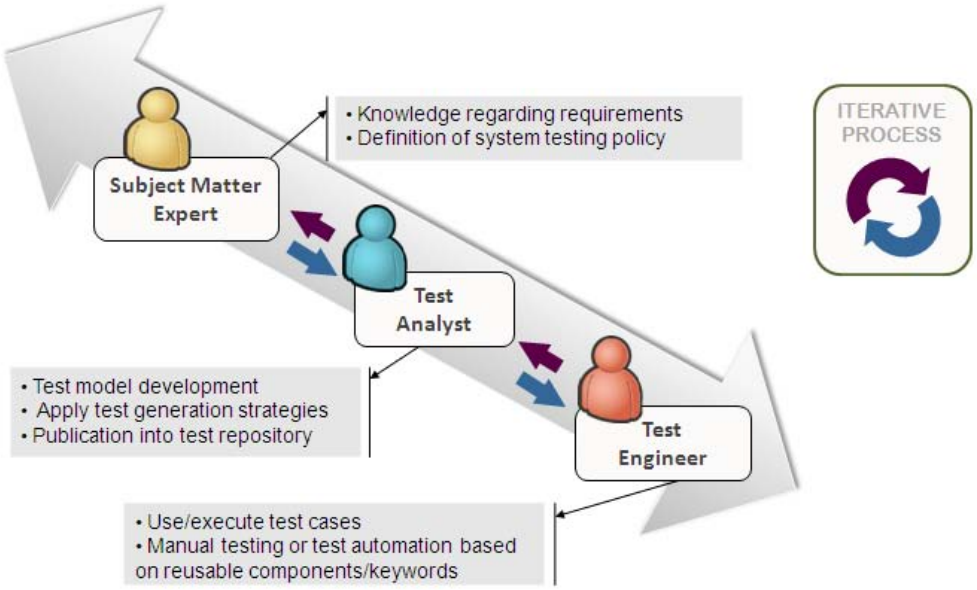

Figure 3. Main roles in the MBT process.

1. The **Test Analyst** interacts with the customers and subject matter experts regarding the requirements to be covered, and then develops the test model. He/she then uses the test generation tool to automatically generate tests and produce a repository of test suites that will satisfy the project test objectives.

2. The **Subject Matter Expert** is the reference person for the SUT requirements and business needs, and dialogues with the test analyst to clarify the specifications and testing needs.
3. The **Test Engineer** is responsible for connecting the generated tests to the system under test so that the tests can be executed automatically. The input for the test engineer is the test repository generated automatically by the Test Analyst from the test model.

The test analyst is responsible of the quality of the test repository in terms of coverage of the requirements and fault detection capability. So the quality of his/her interaction with the subject matter expert is crucial. In the other direction, the test analyst interacts with the test engineer to facilitate test automation (implementation of key-words). This interaction process is highly iterative.

Testing nature and levels

MBT is mainly used for functional black-box testing. This is a kind of back-to-back testing approach, where the SUT is tested against the test model, and any differences in behavior are reported as test failures. The model formalizes the functional requirements, representing the expected behavior at a given level of abstraction.

Models can also be used for encoding non-functional requirements such as performance or ergonomics, but this is currently a subject of research in the MBT area. However, security requirements can typically be tested using standard MBT techniques for functional behavior.

Regarding the testing level, the current mainstream focus of MBT practice is system testing and acceptance testing, rather than unit or module testing. Integration testing is considered at the level of integration of subsystems. In the case of a large chain of systems, MBT may address test generation of detailed test suites for each sub-system, and manage end-to-end testing for the whole chain.

Example: application on actiTIME

Now, we illustrate a full MBT process on a typical web application. This time tracking application, named actiTIME, is freely available on the web (www.actitime.com). In this section, we use this application to demonstrate the various steps of deploying the MBT process.

We illustrate it with Test Designer from Smartesting, which is a model-based testing solution dedicated to enterprise IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite. Test cases are generated from a behavior model of the SUT, using requirements coverage and custom scenarios as test selection criteria. Test Designer models are written in a subset of standard UML.

Test Designer supports both manual and automated test execution, using an offline approach. The generated test cases can be output to test management systems like HP Quality Center, IBM Rational Quality Manager or the open-source tool TestLink, with bidirectional traceability and full change management for evolving requirements.

actiTIME overview

actiTIME is a time management program developed by Actimind. Details about its features, and free downloads, can be found on the website www.actitime.com. Ordinary users have access to their

time track for input, review and corrections. They can also manage projects and tasks, do some reporting and of course they can manage their account (see (1) in Figure 4).

In our sample model we focus on the user time-tracking features of actiTIME version 1.5; after logging into the system the user can specify how much time he spent on a specific task. A typical scenario is as follows:

1. access a time-track;
2. display the time-entry form;
3. type in the hours spent on assigned tasks;
4. the system warns the user that modifications are not saved yet;
5. save the modifications;
6. in case of overtime, the system displays an error message.

actiTIME requirements

In actiTIME a user may have administrator rights. Only administrators can add and remove projects. For a specific project, a user can add or remove tasks, enter the number of hours they spent on a task, etc. To precisely define the expected functional requirements of the actiTIME feature that we model, a list of requirements is defined in Table 1. Summary of actiTIME requirements.Table 1. The IDs are useful as they allow you to see in the test repository which requirements are covered by each of the generated tests.

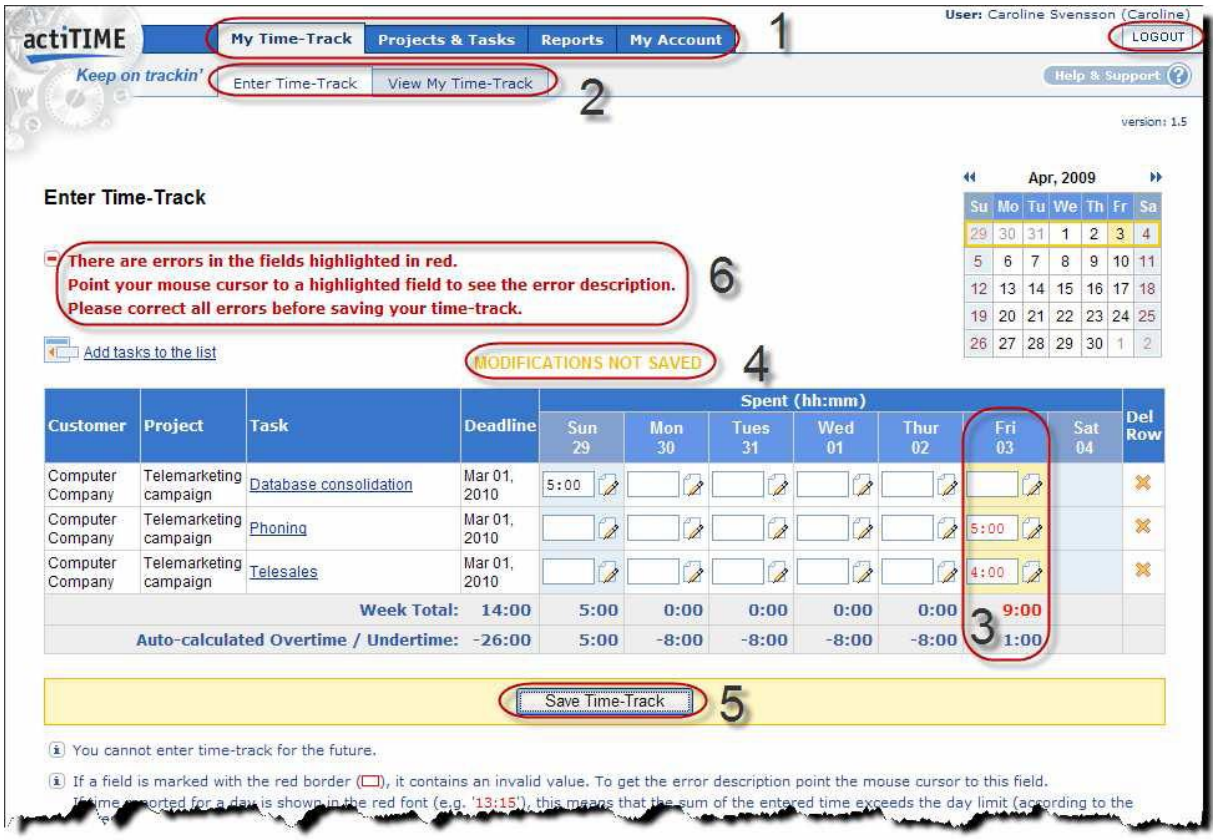

Figure 4. actiTIME user interface.

Requirement Id	Requirement description
ADMIN/ADD_PROJECT	An administrator can add a new project into the system. A project is linked to a customer and includes several tasks.
ADMIN/DELETE_PROJECT	An administrator can delete a project from the system.
LOGIN	When the user try to log on with an incorrect username or password an error message is displayed.
USER/VIEW TIME-TRACK	A user can display its time-track for the current week or any week, in order to report its activity
USER/ENTER_TIME	A user can enter the number of hours spent on his assigned tasks for one or several days.
USER/REMOVE_TIME	A user can correct the number of hours spent on a task by removing some time.
USER/SAVE_TIME	After modifying its time-track, the user can save the changes
USER/SHOW_TIME_TRACKING	A user can display its time-track consolidation for any month.
USER/WORKING_TASK	A user can add or remove task from the task list.

Table 1. Summary of actiTIME requirements.

actiTIME test model

The test model represents the expected behavior of the application, covering the requirements of Table 1. It is based on three UML diagrams (see Figure , Figure and Figure):

- the class diagram represents the business entities and the user actions to be tested;
- the layered state machine represents the dynamic expected behavior;
- the instance diagram gives some test data and initial configuration of the application.

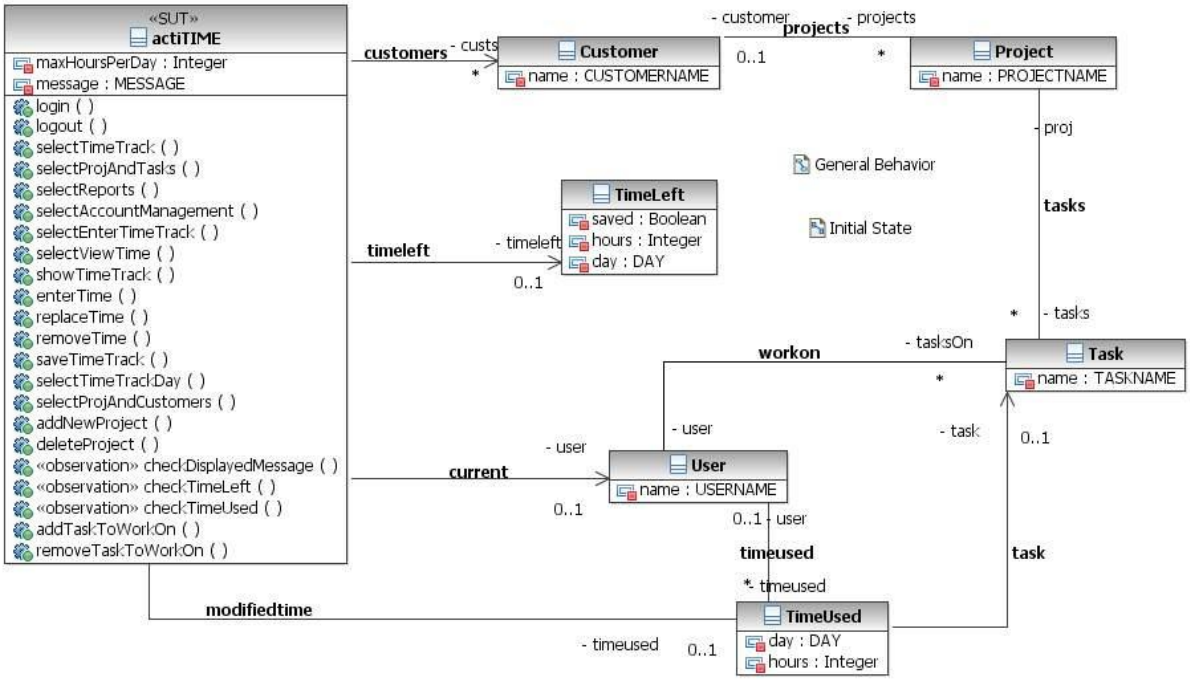

Figure 5. Class diagram for actiTIME test model.

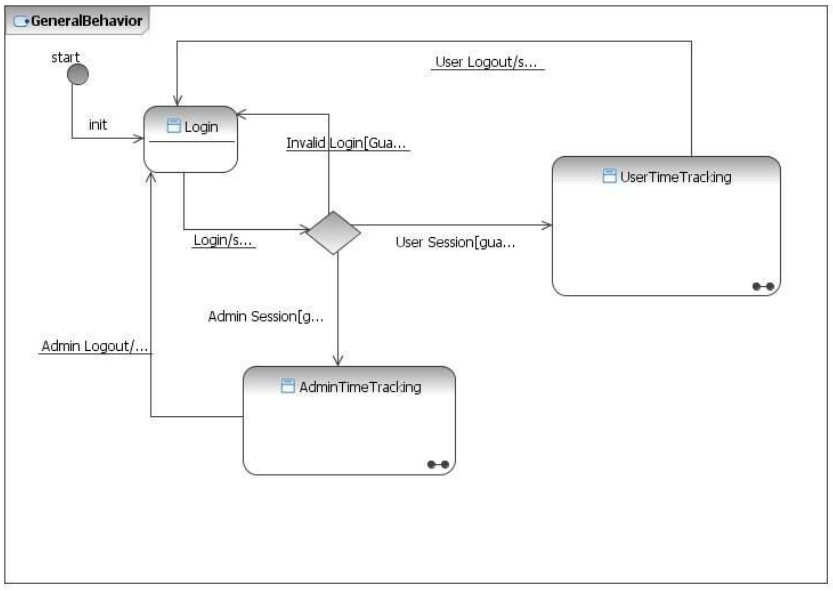

Figure 6. High level state machine for actiTIME (partial).

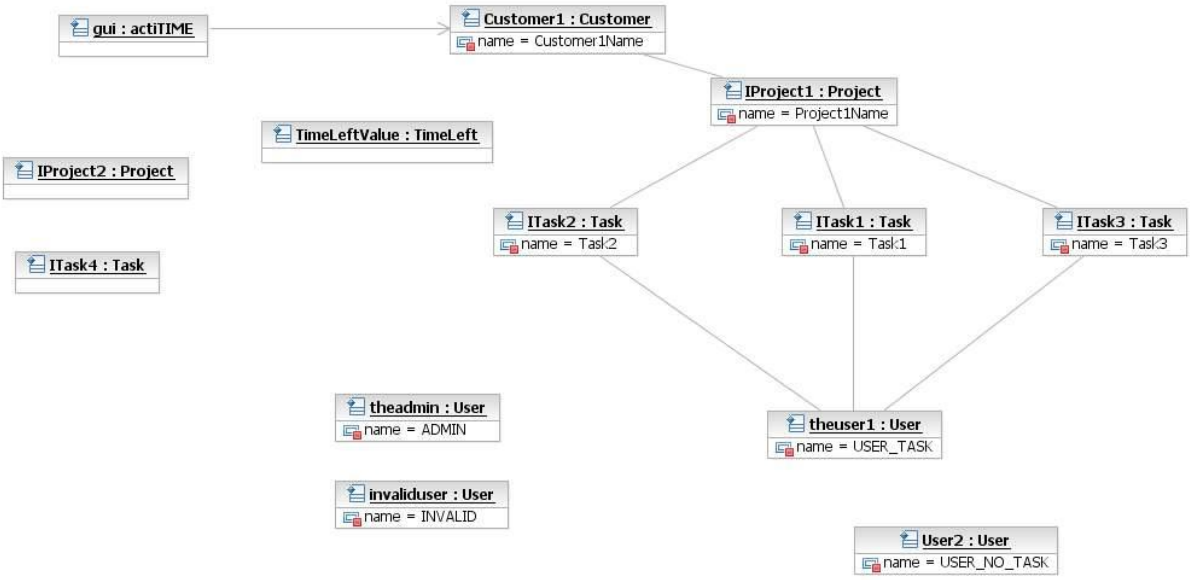

Figure 7. Object diagram for actiTIME.

Figure gives an OCL specification describing the Login operation with an invalid user name. Notice how the requirements are linked to the specification using annotations (---@Req: LOGIN). The annotation ---@AIM gives more detail about which part of that refinement is being modeled here.

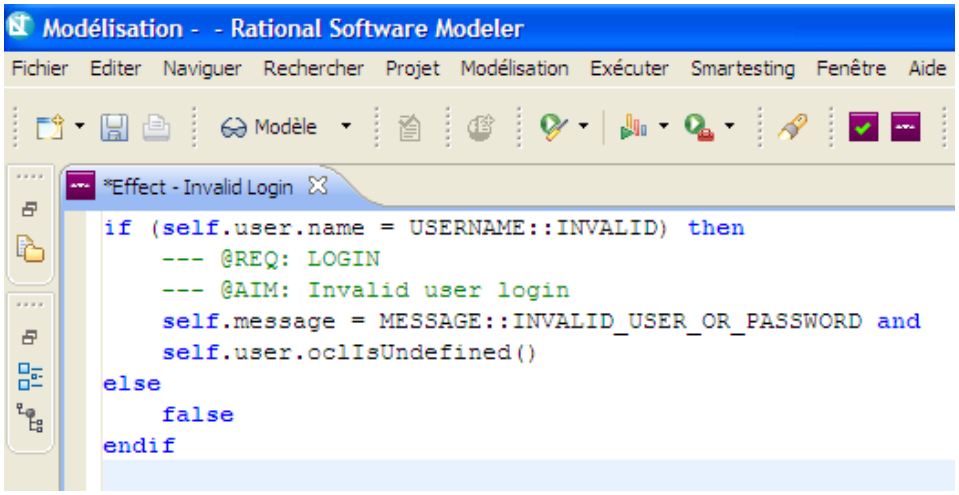

Figure 8. OCL specification for Login (the invalid login case).

Test generation with Test Designer

Figure shows the GUI of Test Designer for the project actiTIME. A list of the generated test cases (structured by test Suites) is displayed on the left, and the details of one test case are displayed on the right. The details of the requirements and test aims that are covered by a particular test step are shown in the right-hand bottom corner.

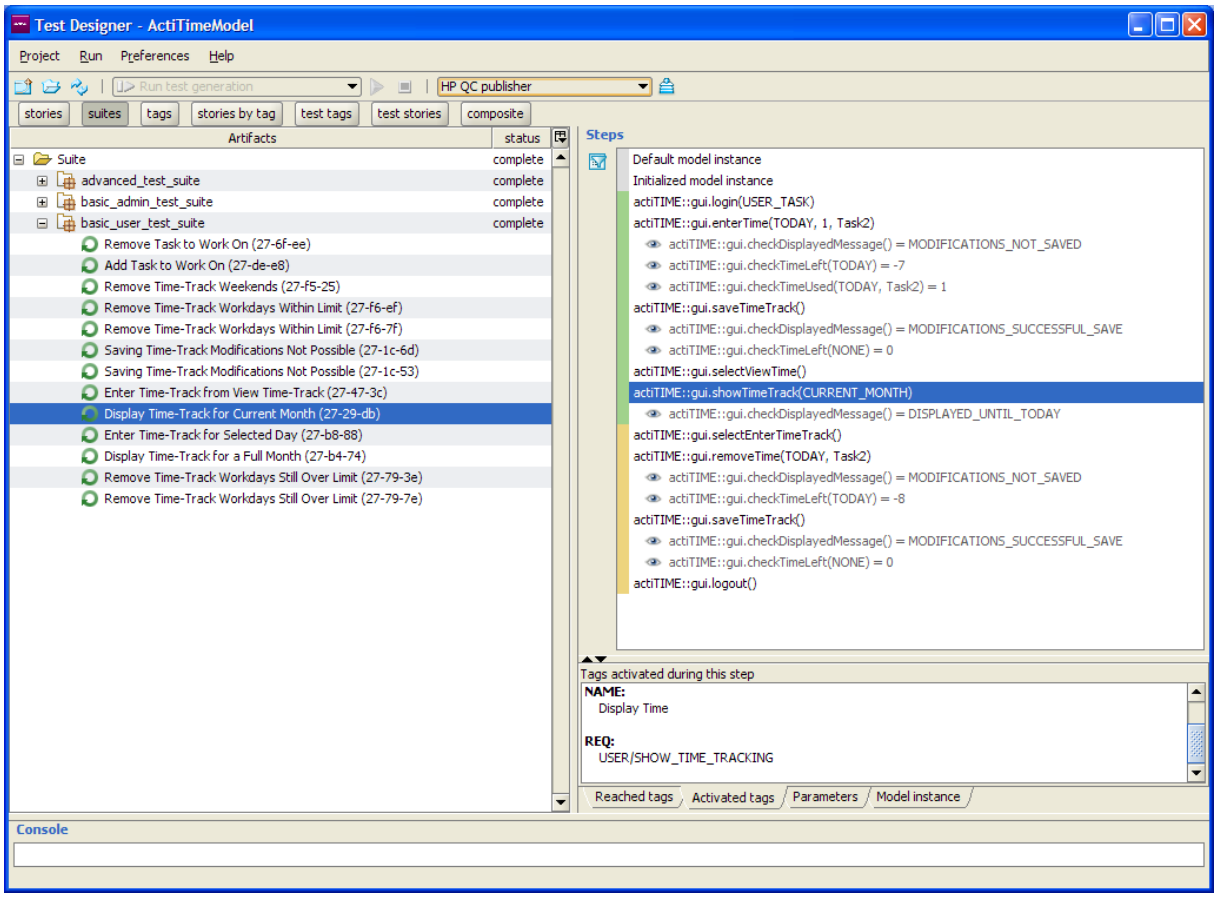

Figure 9. Smartesting Test Designer user interface. Project actiTIME.

Figure 10 shows the generated tests published into a test repository (in this case: HP Quality Center). These tests are ready for manual test execution. Each test is fully documented in the Design Steps Panel.

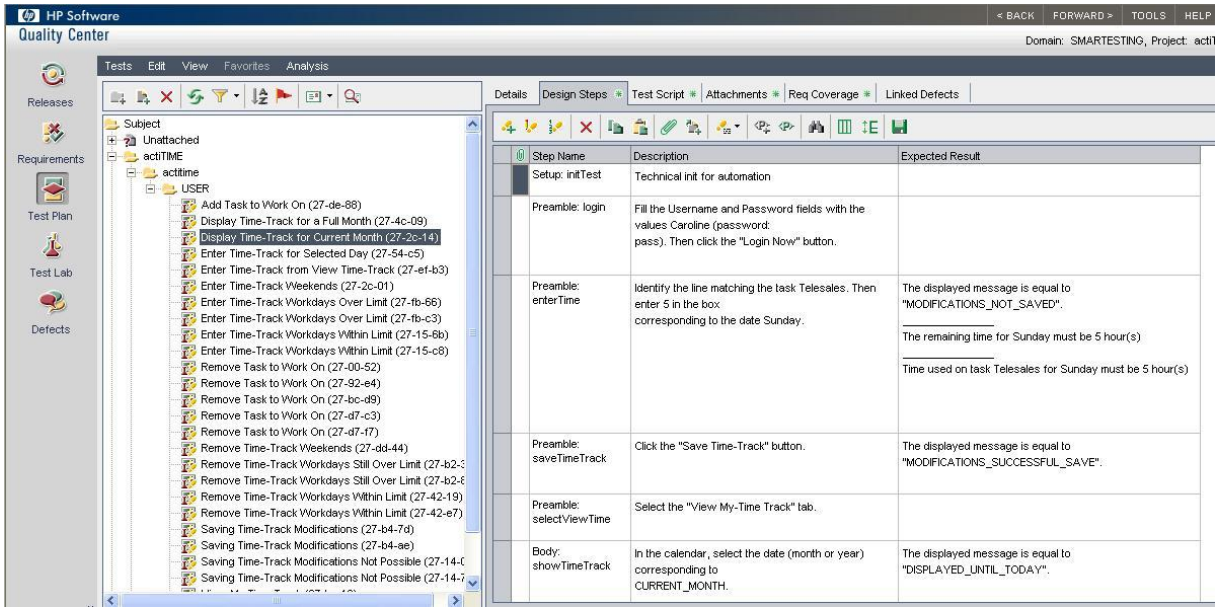

Figure 10. Publication of generated tests into the test manager environment (HP Quality Center)

For test automation, complete script code is generated and maintained for each test case (see Figure 11). The remaining (optional) task for the test automation engineer is to implement each key-word used in UML test model so that it is defined as a sequence of lower-level SUT actions. If this is done, the generated test scripts can be executed automatically on the SUT. An alternative approach is to leave the key-words undefined, in which case a human tester must execute the scripts manually.

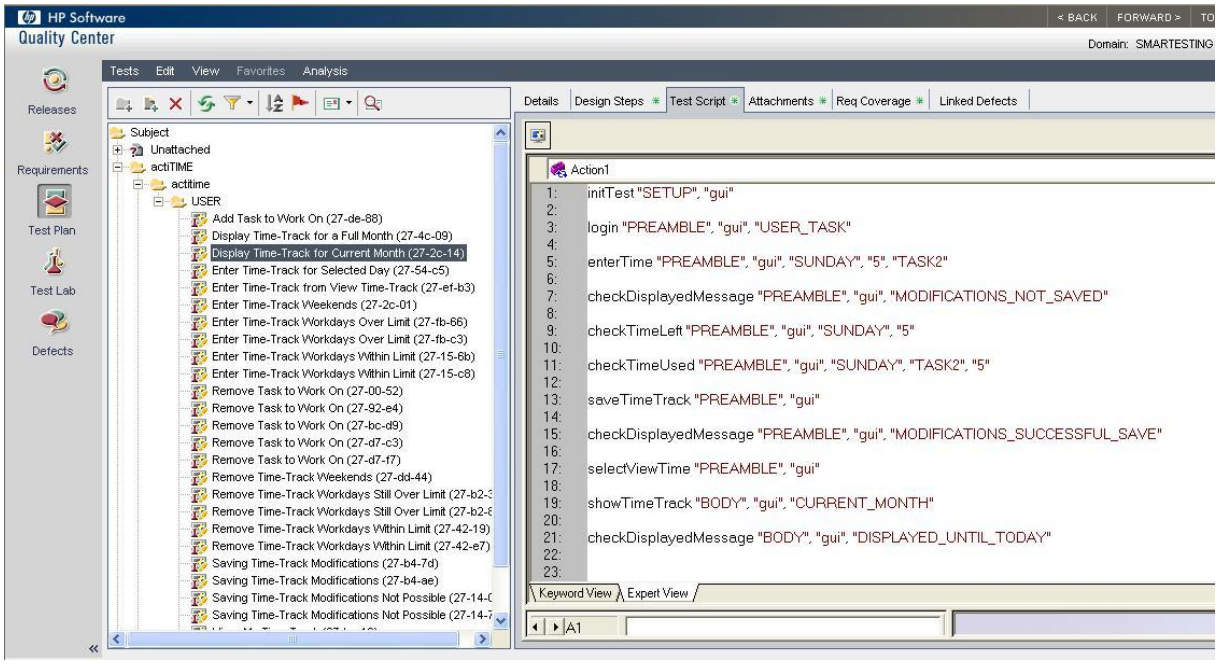

Figure 11. Publication of generated scripts into the test manager environment (HP Quality Center)

To sum-up, we deployed on the actiTIME application a typical MBT solution for IT applications, using a subset of UML as input language (class diagrams, state diagrams, instance diagrams, and OCL specification language), providing automated test generation and publication features both for manual and automated testing.

Key factors for success when deploying MBT

Here we describe the keys to success when deploying a MBT approach and tools. The key factors for effective use of MBT are the choice of MBT methods that are used, the organization of the team, the qualification of the people involved, and a mature tool-chain. Then, you may obtain the significant benefits that we discuss at the end of this section.

1. **MBT Methods, requirements and risks**: MBT is built on top of current best practices in functional software testing. It is important that the SUT requirements must be clearly defined, so that the test model can be designed from those requirements, and the product risks should be well understood, so that they can be used to drive the MBT test generation.
2. **Organization of the test team**: MBT is a vector for testing industrialization, to improve effectiveness and productivity. This means that the roles (for example between the test analyst who designs the test model, and the test engineer who implements the adaptation layer) are reinforced.
3. **Team Qualification - test team professionalism**: The qualification of the test team is an important pre-requisite. The test analysts and the test engineers and testers should be

professional, and have been given appropriate training in MBT techniques, processes and tools.

4. **The MBT tool chain**: This professional efficient testing team should use an integrated tool chain, including a MBT test generator integrated with the test management environment and the test automation tool.

Expected benefits

Model-based Testing is an innovative and high-value approach compared to more conventional functional testing approaches. The main expected benefits of MBT may be summarized as follows:

- Contribution to the quality of functional requirements:
  - Modeling for test generation is a powerful means for the detection of "holes" in the specification (undefined or ambiguous behavior) ;
  - The test phase may start earlier and find more flaws in the requirements repository than "manual" test design approach.
- Contribution to test generation and testing coverage:
  - Automated generation of test cases;
  - Systematic coverage of functional behavior;
  - Automated generation and maintenance of the requirement coverage matrix;
  - Continuity of methodology (from requirements analysis to test generation).
- Contribution to test automation:
  - Definition of action words (UML model operations) used in different scripts;
  - Test script generation;
  - Generation of the patterns for automation function library;
  - Independence from the test execution robot.

Conclusion

The idea of model-based testing is to use an explicit abstract model of a SUT and its environment to automatically derive tests for the SUT: the behavior of the model of the SUT is interpreted as the intended behavior of the SUT. The technology of automated model-based test case generation has matured to the point where the large-scale deployments of this technology are becoming commonplace. The prerequisites for success, such as qualification of the test team, integrated tool chain availability and methods, are now identified, and a wide range of commercial and open-source tools are available.

Although MBT will not solve all testing problems, it is an important and useful technique, which brings significant progress over the state of the practice for functional software testing effectiveness, increasing productivity and improving functional coverage.

References

[1] M. Utting, B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffman, 2007.

[2] Hung Q. Nguyen, Michael Hackett, and Brent K. Whitlock, *Global Software Test Automation: A Discussion of Software Testing for Executives*, Happy About, 2006.

[3] Elfriede Dustin, Thom Garrett, and Bernie Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, Addison-Wesley Professional, 2009).

About the author

Dr. Bruno Legeard is Chief Technology Officer of Smartesting, a company dedicated to model-based testing technologies and Professor of Software Engineering at the University of Franche-Comté (France).

In 2007, Bruno Legeard authored with Dr. Mark Utting the first industry-oriented book on model-based testing, "Practical Model-Based Testing: A Tools Approach", Morgan & Kaufmann Publisher.

He started working on model-based testing in the mid 1990's and has extensive experience in applying model-based testing to large information systems, e-transaction applications and embedded software.