Pratikte Model Tabanlı Test: Bankacılık Alanından Bir Deneyim Raporu

Şerafettin Şentürk1, Abdurrahman Akın1, Ayşe Betül Karagöz1, Vahid Garousi2
Kuveyt Türk R&D Center Kocaeli, Türkiye1
{serafettin.senturk, abdurrahman.akin, ayse.karagoz}@kuveytturk.com.tr
Queen’s Üniversitesi Belfast Belfast, Birleşik Krallık2
v.garousi@qub.ac.uk


Anahtar Kelimeler: Model Tabanlı Test, Graphwalker, İnternet Bankacılığı, Deneyim Raporu, Endüstriyel Vaka Çalışması

Model Based Testing In Practice: An Experience Report From The Banking Domain

Şerafettin Şentürk1, Abdurrahman Akın1, Ayşe Betül Karagöz1, Vahid Garousi2
Kuveyt Türk R&D Center Kocaeli, Turkey1
{serafettin.senturk, abdurrahman.akin, ayse.karagoz}@kuveytturk.com.tr
Queen’s University Belfast Belfast, UK2
v.garousi@qub.ac.uk

Abstract. Model-driven software engineering has become more popular in recent years. Due to the high number and diversity of users, new testing approaches are necessary to reduce the occurrence of faults and ensure higher quality in industrial applications. The objective of this paper is to evaluate the use of Model-Based Testing (MBT) practices in the development and execution of automated test suites to verify and validate internet-banking solutions in the context of a large banking institution in Turkey.

Keywords: Model Based Testing, Graphwalker, Internet Banking, Experience Report, Industrial Case Study

1 Introduction

Model-based techniques are also heavily used in testing phase of the software development life cycle. Model-based testing (MBT) is a type of testing that relies on explicit behaviour models that express the intended behaviours of a SUT and/or the behaviour of its environment. Test cases are generated from one of these models or their combination, and then executed on the SUT. The ideas of model-based testing, then dubbed
pecification-based testing, date back to the Seventies. Recent emphasis on model-based and test-centered development methodologies as well as the level of maturity of technology from the area of formal verification have led to a strong increased interest in the subject in the past decade, both in the academic field and in the industry. Model-based testing encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases [1].

2 Industrial Contexts And Need Analysis

2.1 Industrial context and a review of the System Under Test (SUT)

Kuveyt Türk Participation Bank Inc. is a private financial (banking) institution in Turkey. The bank started its operations in the Turkish finance market in 1989. It has 414 branch locations across Turkey, and delivers a wide range of financial products and services to customers. As of year-end 2016, the bank employed about 6,000 personnel [8].

The SUT running on the bank side is the main web application used by the customers. It is called Internet Şube (in Turkish), meaning Internet Branch, and provides user interface in Turkish and English. The app provides typical features for personal and corporate banking, e.g., transfer funds, view financial reports, and access to different bank cards owned by customers. In terms of codebase size, the app is about 293 KLOC. The app has 162 GUI screens (pages) in total, one of them is shown in Figure 1, which is the app’s main window. As of this writing, the app has about 180,000 active users, and a monthly traffic of about 1.6 million logins.

![Fig. 1. Screenshot of the SUT](image-url)

2.2 Need analysis and motivations for the study

The web application was first released in 2007 for the retail and corporate customers. Furthermore, since its earliest version, the app has been undergoing all four types of software main-
tenance activities [9]: adaptive, perfective, corrective and preventive maintenance. Adaptive maintenance has been conducted since its first version to modify the system to cope with changes in the software environment (e.g., DBMS). Perfective maintenance has been conducted for implementing new or changed features. Corrective maintenance has dealt with diagnosing and fixing defects. Last but not the least, preventive maintenance has been done for increasing software maintainability, reliability, and to prevent problems in the future, e.g., using practices such as refactoring. Hence each maintenance activity on the SUT has made testing activities necessary. It goes without saying that, given the criticality of the app, its correct and uninterrupted functionality is of utmost importance. Test case generation was conducted manually, which had various drawbacks, e.g., taking too much human resources, being error prone and missing defects. Requirements are the bottom line of any application under development. MBT (Model-based testing) is very useful especially when dealing with requirements. In MBT, models are developed from the requirements for the system. The model is used to generate test cases, typically in an automated fashion. The system is run against the generated tests and the outputs are compared with the expected output. In our current SUT, requirements are written in an unstructured format and without a formal flow diagram. There was a need for strong collaboration and unambiguous develop/test base documentation between the different stakeholders: business stakeholders, system analysts, developers and software testers. Models are a good example of a visual and formal documentation. With model-based testing, requirements can be documented more systematically and showed as an overview to the stakeholders. Another advantage of model-based testing is that test case generation can be done automatically, otherwise test cases has to be gained and written manually from unstructured rich text requirements. On the other hand, by means of MBT, Maintenance of test cases will be decreased, because automatic test case generation can be done after changes in the requirement/model. From this perspective it is worth to mention about two main approaches for test execution in MBT: Online and Offline. Both approaches have its usages depending on the testing framework. In offline MBT, generation of test case is automated. Online MBT test case generation and execution are in motion.

3 Related Work

In Dalal, Siddhartha R., et al. [2] the report on various practices of development of tools and various methodologies related to model-based testing was generated. Some case studies have been shown which provides details and results of application of combination of test-generation methods on a large scale to diverse applications. Based on these, an insight has been given into what is practiced and what are the obstacles in transferring these technology to testing organizations. Lyu, M.R [3] offers a view of the development, testing, and evaluation schemes for software reliability, and its integration to form a unified and consistent paradigm.

Practical model-based testing: a tools approach [4] gives an insight to model-based testing in a practical manner, shows the way to write models for testing purposes and usage of model-based testing tools to generate test suites. It aims at testers and software developers who wish to use model-based testing, rather than at tool developers or academics used in specification based testing. In Fujiwara, S.et al. [5] methods for the selection of appropriate test case, an important issue line with testing of protocol implementations as well as software engineering, is shown.

Sing, Vasudha and Ramasamy, Subburaj have done a study of model-based testing. In this study firstly MBT explained, and basic MBT approach and types of model in MBT approach have been shown. Two case studies are given in their paper which will help to understand the basic concept of MBT and also describe the way to generate test suits from the model. They created their MBT model for the web application. In their
case study they have basically six pages like hotels, flights, activities, packages, cars, cruises and for each page there is individual form. Their proposed study is primarily for verifying the functionality of the SUT. And then the study showed that MBT provides the roadmap for automatic testing of software. It provides effective test strategies without missing any important test cases. Model-based testing is an efficient and adaptable method of testing software by creating a model describing the behaviour of the system under test. The availability of tools has particularly makes the job easier [6].

4 Research Design And Setup

Based on our research approach as a type of action research, after identifying the problem and needs of the project, we defined the problem as a case study that we discuss its design aspects later.

In terms of the research approach, given the nature of our project and the context, we designed an exploratory/ improving case study [10] to be executed in an observational uncontrolled context.

Stated using the GQM’s goal template [20], we formulated the problem (goal) of the project as follows: the goal of the project was to develop a set of end-to-end model-based testing test suites that would cover the most important requirements of our SUT in a formalized way and generate test cases automatically from them. And finally connected the generated test cases with the test code scripts to create a chain of fully automated and ready to execute system. Based on the above goal, we posed the study’s research question (RQ) as follows: What are the costs, benefits and ROI of automated testing in this industrial context? To achieve this, we conducted a case study for cost-benefit analysis of the automated test suites in the specific industrial context, from the point of view of the stakeholders involved in the project.

Given the complex nature of the project we planned a rigorous empirical approach to address the study’s RQ by adopting heuristics from the literature and our recent works in this area.

5 Test Automation: Strategy, Approach, Development Of Models

In this section, we present the details of the automated testing project, including the approach, the test suites that we developed and the best practices that we applied.

5.1 An Overview of Our Test Automation Strategy

To ensure success in test automation, it is important to define and follow a proper test automation strategy [11]. Automating every case and executing every case is not possible. There is limited time and resource. We are essentially evaluating the “when to automate” question for our test automation decisions in each step, using our past experience in this area [12-14].

Automated execution is done for the regression tests, because these tests are executed frequently and on different platforms in parallel. We are also familiar with other best practice and test patterns in the area of software test-code engineering and utilized them when they were suitable for the test method under development. For example, we developed test-specific utility libraries containing utility functions which are used by multiple test methods, decoupled (separated) test data from test logic, separating the locators. In this action research we tried to automate the test case generation and execution. Some aspects of our test automation strategy are: selecting the right test cases, selecting the right test automation tool, reusability of test automation code, execution
and maintenance of test cases. Together with our stakeholders, we selected the most important/critical cases (login and money transfer cases) with the highest risks. The likelihood, impact and usage are used as a risk criteria.

5.2 Choosing the Right Test Automation Tools

An important decision in the test automation strategy is the choice of the “right” test automation tool(s) [15]. The basic idea in model-based testing is generation of models which can be transition system, UML State Machines, finite state machines, class diagrams along with constraints, etc. [7-12] from which complete test cases which is input and expected output pairs that can be generated. In online testing the model and the considered system for test are connected and the tests run dynamically whereas in offline testing the test cases/suites are generated which can be later tested with the system using a tool. A model which describes the system or software under the testing process can be taken as its partial or abstract behaviour towards the system. Hence the test cases which are taken out from a model can be said as functional tests on the similar platform of abstraction as the model. These test cases are collectively known as an abstract test suite.

Most of the available MBT tools support mainly automatic test case generation rather than test data generation and test script generation due to two reasons: firstly test case generation requires complicated strategies involving various test selection criteria from MBT models, and the generation results heavily rely on the selected criteria and strategies; secondly test case generation brings much more testing benefits, as the main efforts spent on traditional testing lies on manual preparation of test cases.

There are several tools which can be used for model-based testing: Conformiq designer commercial tool in which models can be created as UML State Machines and in Qtronic Modeling Language (QML). Tests can be exported to test management tools or TTCN-3.

Spec Explorer which develops the programs in C# is a commercial tool is a successor of AsmL test tool which is now integrated with Visual Studio.

TestOptimal Tool supports FSM and E-FSM modelling with several test case generation algorithms. It has various plugins for online testing web application, windows applications, database and web services, etc. It also supports data-driven testing and pair-wise algorithms right within the model and has the facility for performing load testing using the same models.

ModelJUnit Tool allows you to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics.

TransWare BPM-X Tool creates test cases from business process models based on different criteria (statement, branch, path, condition). It can import models from several modelling tools, and can export test cases to Excel, HP Quality Center, etc.

VERA Tools is an academic tool for vulnerability testing, which allows users to define attacker models by means of extended finite state machines, and correspondently generates test cases targeting generic vulnerabilities of Web applications. In order to efficiently perform tests, VERA also provides a library containing common vulnerability test patterns for modelling [16].

MoMuT Tool is a set of model-based test case generation tools that work with UML state machine, timed automata, requirement interfaces and action systems. This tool features a fault-based test case generation strategy that allows mutations to be made on models and generates richer test cases from both original and mutated models to detect if models contain certain user-selectable or seeded faults. A fault localization mechanism is included in MoMuT for debug when a test case fails [16].

MISTA Tool is an open-source tool that generates test cases from models of finite state machine or function net. Both control-oriented and data-oriented models can be built by MISTA. The formats of test cases cover a number of languages (Java, C, C++, C#, PHP, Python, HTML and
VB) and test frameworks (xUnit, Selenium IDE and Robot framework). MISTA supports both online and offline testing [7].

GraphWalker which is an open source tool in which test cases are made from finite state machines and uses search algorithms to a-star (generating the shortest path to a specific vertex or edge) or random search algorithm to cover various states, edges, requirements. By using Grapwalker, the system model and requirements can be easily modelled in a graph based components like edges and vertices.

After our research we decided to choose GraphWalker, because it is open-source, has complete documentation with examples and is easy to learn and use. It uses directed edges and vertices to automatically create the paths that corresponds to SUT Test Cases. Another significant reason to select the GraphWalker is the Selenium integration to run the test cases automatically for our web based internet banking.

GraphWalker is generally speaking a tool for generating offline and online test sequences from Finite State Machines. In order for MBT to understand the model; there are some simple rules to follow. Every graph must have one (and only one) vertex with the label 'Start'. This shows MBT the starting point in the graph. Every vertex in a graph must have a name. Edges usually have names, but are not required to have them. The name shall follow whatever naming convention is required of the test execution engine. Labels represent the name of the vertex or edge. Keywords are written within the label of a vertex or an edge. They must exist on their own line in the label, but not the first line. In course of the project, graph editing tool namely yEd is used which is an open source tool used to design a model.

The model in the GraphWalker is built using a finite-state machine [FSM]. A FSM (Finite-State Machine) is a model with vertices (nodes) and edges (arrows) which connects the vertices with each other. There are certain set of standards which GraphWalker uses:

- A model consists of vertices and edges.
- A vertex represents a state in the system under test.
- A vertex must have a defined label.
- You can have multiple vertices with the same name.
- An edge represents the transition between two states.
- An edge can have a have a label, but it is not mandatory.
- A model must have one and only one vertex with the label Start.
- The Start vertex must have one and only one out- edge which has a mandatory label.
- The Start vertex must have no in-edges.

An edge is a transition which can be a click on a button, a timeout, a server API call etc. The label of an edge will map against a method, function or sub routine during test execution. Best practice is to start the name with 'e_'. The reason is that it is easier to recognize the function in the test execution tool, as an edge. The same name can be reused in the model. A vertex is some kind of state that is possible to verify using some kind of oracle. A vertex must always have a label. The label of a vertex is mapped against a method, function or sub routine during test execution. Best practice is to start the name with 'v_'. The reason is that it is easier to recognize the function in the test execution tool, as a vertex.
5.3 How the Test Models are Designed

![Diagram of the Login Page Model and Test Case Generation]

Our models are function oriented, simply designed without having complex models. We discussed and thought about how to model the different functionalities of our internet banking application. After these discussions we designed and reviewed the models. We tried to use small models with shared vertices to combine the models. Edges are the actions like clicks, sending input to text fields and vertices are the validations on the pages after the actions. An example is given in Figure 2: e_CloseSecurityPopup will close the popup with a click and v_LoginPage will validate whether we are on the right page. We tried to add all critical/important positive paths to the model and some important negative paths like e_EnterInvalidCredentials.

After saving the models we generated the test cases by the help of GraphWalker as paths, see Figure 2. Login is our base model for the whole internet banking system, because in order to
trigger a transaction in our internet banking application, any customer has to be logged in to the system. In the modelled system, since dashboard page is the main page referenced from other pages, it is labelled as SHARED. v_DashboardPage SHARED:DashboardPage is a shared vertex and combines the paths.

Fig. 3. Login Page Model and Test Case Generation

Fig. 4. Money Transfer Model
For each node and vertex we implemented test automation with Java and Selenium. This made it possible to automatically execute the generated paths. Edges starting with e_ were implemented as actions and vertexes starting with v, see Figure 4 e_EnterInvalidCredentials and v_LoginInvalid Credentials Page.

As one of the critical functions of internet banking is money transfer, money transfer models have been designed that can be seen in Figure 3. For money transfer function two main subfunctions namely money transfer to same bank account and money transfer to IBAN account use cases have been modelled.

In addition to this, Currency Exchange models, as Currency Buying and Currency Selling cases have been designed and corresponding Java Selenium Codes have been written.

```java
public void e_EnterInvalidCredentials()
{
    methodsPage.setObjectBy(USERNAME, customerNo);
    methodsPage.setObjectBy(PASSWORD, wrongPassword);
    methodsPage.clickObjectBy(SUBMITBTN);
}

public void v_LoginInvalidCredentialsPage()
{
    String errorMessage = driver.findElement(By.cssSelector("div.inerAreaErrorShort > p")).getText();
    Assert.assertEquals(expectedErrorMessage, errorMessage);
}
```

Fig. 5. Sample Selenium Codes for Login Model

The generic actions are collected on a Library page named methodsPage to make it reusable, see Figure 5. The same is done for the different locators to increase the maintainability, see Figure 5.

```java
protected WebElement setObjectBy(By by, String value)
{
    WebElement element = driver.findElement(by);
    element.clear;
    element.sendKeys(value);
    return element;
}

protected WebElement clickObjectBy(By by)
{
    WebElement element = driver.findElement(by);
   waiter.until(ExpectedConditions.presenceOfElementLocated(by)).click();
    return element;
}

public WebElement getElementBy(By by)
{
    return driver.findElement(by);
}
```

Fig. 6. Reusable methods and variables declaration

```java
By USERNAME = By.id("intUsername");
By PASSWORD = By.id("Password");
By SUBMITBTN = By.id("btnSubmit");
By LOGIN = By.id("LoginButton");
```

Fig. 7. Reusable methods and variables declaration
6 Evaluation of The Approach

In terms of quantitative benefits the following improvements were noticed: time to create the test basis, test base maintenance, development time, time to generate the test cases, test case maintenance.

As also for quantitative benefits, Time to maintain the test basis, time to generate and maintain the test cases is decreased.

Shown in Table 2 time to create the test basis did not change but other benefits are better on the model based testing side. Time to generate the test cases decreased with 50% compared to manual test case writing. As shown as Table 2 time is 2 days for to generate the test cases before model based and then time is 1 day after model based. Especially test case maintenance is reduced, because test cases can automatically generated. Test case maintenance time fallen from 1 day to 2 hours as shown in the Table 2. When there is a change applied in the test basis (changed requirement) test cases has to be maintained. In the traditional way a software test engineer has to read the changes and maintain the test cases, but with MBT test cases can be generated automatically. This means that test case maintenance is unnecessary. So testing can be done faster and more reliable. This provides to decrease human effort to create the test cases.

In terms of qualitative and intangible benefits the following improvements were noticed: testability, close collaboration among stakeholders (business stakeholders, analysts, developers, testers), ambiguity of requirements, human effort to create and maintain the test cases. A test basis of tens of pages can be shows in one model which has a clear overview of the functionality. This provides also easy maintenance of the test basis, developers detect changed requirement so development time decreases and automated test’s testability increases.

Shown in Table 2 all qualitative and intangible benefits are changed positively. In general MBT approach provided faster software development life cycle with less effort. All evaluation are summarized in the Table 2 that it shows as numeric values and ratings.

7 Discussions

In our test models like login and money transfer, we tried to keep our models simple and easy to understand. From this perspective we made our system models modular and reusable. We connected the modular models together with SHARED attribute and tried to avoid developing big complex models. Our internet banking web application has about 178 functionalities and modeling it in one model would make it complex and unreadable. Giving the models to GraphWalker we automatically generated the test cases. These test cases can be executed manually, but we automated the execution with Java-Selenium.

Each vertex in our model is a validation and we tried to validate only the most critical points on the page. Developing a strategy is important for model based testing, otherwise adding each detail in one model will make the model complex and unreadable. This is also the case for writing the automation code, validating each component in a page can fail an execution because of insignificant details. Too detailed models will result in more test cases, more test cases will lead to more test automation code and these will all cause the increase in execution time.

In our test models, we have firstly generated the test cases with the random 100% edge coverage and GraphWalker generated some redundant steps. Redundant test cases cause an increase in the number of test cases and test execution time. By taking advantage of quick-random generator feature in GraphWalker we could have chance to result in less redundant steps.
8 Conclusions And Future Work

Models are very important in model based testing. They have to be complete and well organized. Automated test case generation is the most important benefit of model based testing and also we can see the whole picture of the SUT shown in the model. On the other hand, collaboration with the stakeholders about the SUT has improved in terms of the clear overview in the model.

For generating the test cases we used random generator algorithms. As future work, trying different path generation algorithms which are supported by GraphWalker can be quite challenging and the results can be analysed within each other. As a result of comparison of different path generation algorithms, the number of generated test cases and test execution time can be decreased.

GraphWalker will release “shortest_all_paths” generator algorithm and this algorithm will calculate and generate the shortest path through the model. This generator is not available yet, so we could not try it. As a future work when this feature is available, it could be tried for further detailed analysis of the paths of the graph.

Another interesting future work can be the usage of weights and guards in our models. Using these futures we can guide the generation of test case steps. Weights are used only when using the random generator. It holds a real value between 0 and 1, and represents the probability that the edge should be chosen. Guards are a mechanism only associated with edges. Their roles are the same as an if-statement, and makes an edge eligible or not for being walked.

Table 1. Summarizing the benefits of model based testing

<table>
<thead>
<tr>
<th>Types of benefits</th>
<th>Aspects</th>
<th>Regular manual test based vs model based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before model based</td>
<td>After model based</td>
</tr>
<tr>
<td>Quantitative</td>
<td>Time to create the test basis</td>
<td>4 days</td>
</tr>
<tr>
<td></td>
<td>Test base maintenance</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Development time</td>
<td>8 days</td>
</tr>
<tr>
<td></td>
<td>Time to generate the test cases</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Test case maintenance</td>
<td>1 day</td>
</tr>
<tr>
<td>Qualitative and intangible</td>
<td>Testability</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Close collaboration among business stakeholders, analysts, developers and testers</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Ambiguity of requirements</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Human effort to create the test cases</td>
<td>High</td>
</tr>
</tbody>
</table>

Acknowledgment

This work was in part supported by the European ITEA3 TESTOMAT (the Next Level of Test Automation) project (www.testomatproject.eu) and the Scientific and Technological Research Council of Turkey (TÜBİTAK) via grant #9180076.

References


7. Li Wenbin, Gall Franck Le, Spaseski Naum, A Survey on Model- Based Testing Tools for Test Case Generation, The 4th International Conference on Tools and Methods of Program Analysis, At Moscow, Russia, Volume: 779


