Experiences in Test Automation for Multi-Client System with Social Media Backend

Tuomas Kekkonen, Teemu Kanstrén, Jouni Heikkinen
VTT Technical Research Centre of Finland
Oulu, Finland
{*tuomas.kekkonen, teemu.kanstren, jouni.heikkinen*}*@vtt.fi*

***Abstract*—Effective testing of modern software-intensive systems requires different forms of test automation. This can be implemented using different types of techniques, with different requirements for their application. Each technique has a different cost associated and can address different types of needs and provide its own benefits. In this paper, we describe our experiences in implementing test automation for a multi-client application with a social media backend. As a first option, traditional scripting tools were used to test different aspects of the system. In this case, the test cases were manually defined using an underlying scripting framework to provide a degree of automation for test execution and some abstraction for test description. As a second option, a model-based testing tool was used to generate test cases that could be executed by a test harness. In this case, a generic model of the behaviour was defined at a higher abstraction level and from this large numbers of test cases were automatically generated, which were then executed by a scripting framework. We describe the benefits, costs, and other properties we observed between the two different approaches in our case.**

***Keywords*-model-based testing; test automation; performance testing; data validation testing; web service testing.**

I. INTRODUCTION

Testing is commonly referred to as one of the most time consuming parts of the overall development and maintenance process of a software intensive system. To address this, various techniques have been developed to make test automation more efficient, each with their own costs and benefits. In this paper, we describe our experiences in implementing test automation for a multi-client system with a social-media backend. In implementing this system, two different approaches were applied to create and execute test cases with varying degrees of test automation. Both tested the system through its external interfaces built on top of HTTP requests with JSON data structures (REST style web services). We describe our observed costs, benefits, and limitations of each approach.

The first approach we applied was based on using existing test automation frameworks to provide a scripting platform for manually defining test cases and automating test execution. The tools applied are existing HTTP scripting tools to manually define test sequences for testing specific properties of the system. For us, the benefit with this approach is quick bootstrapping of the test automation process in using off-the-shelf tools, and the ability to manually define specific test cases for specific requirements. The cost is in creating large sets of test cases manually, which quickly becomes labour intensive and exhaustive. The main person responsible for this approach was the first author of this paper.

The second approach we applied was based on model-based testing (MBT). MBT is used to generate test cases from a model describing the system. This model typically describes the behaviour of the system in a form of a state-machine at a suitable abstraction level for generating the required test cases. The MBT tools provide components and modelling notations to make the modelling easier, and algorithms to generate test cases from the models. The test logic and integration with the test setup are domain specific and need to be created separately for each tested system. For us, the benefit with this approach is getting extensive coverage with automated test generation. The cost is in creating the models for test generation and integration with test execution. The main person responsible for this approach was the third author of this paper.

Guidance and coordination for both of these approaches was provided by the second author of this paper.

The rest of the paper is structured as follows. Section II presents the problem domain. Section III presents the manual test setup and experiences. Section IV presents the MBT test setup and experiences. Section V discusses the overall experiences in a broader context. Finally, conclusions end the paper.

II. BACKGROUND

We consider the system under test (SUT) here as a form of a web-application, where the different components communicate over HTTP requests. The service also provides a native mobile client interface and a social-media web-browser interface. However, in the testing phases described in this paper, we were interested mainly in testing the backend service. This is because the concerns of the project parties were on the high bandwidth and data processing requirements set by the data collection and transfer. Thus the user-interface part is not discussed in detail at this point, but only for the relevant interface and data processing parts related to these interfaces.

Often in web application testing the main goal is to get assurance that the service can handle all the user requests without problems. Therefore, it can be seen as performance testing. This requires assessing various properties such as

 ISBN: 978-1-61208-233-2

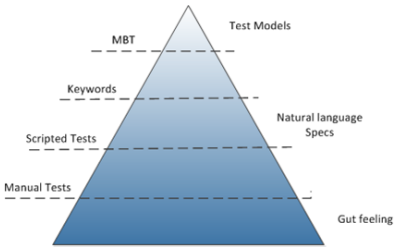

Figure 1. Test Automation Pyramid.

response latency, varying event sequences, event frequency handling, and error handling(as described e.g., in [1]). To estimate the capabilities the tester has to have some idea how many users the service will have and what type of requests there will be. The variance in effect to the server between requests can be in processing load, I/O load and network load. Based on the estimates the service is loaded with different requests and then maybe bottlenecks are discovered. Then the development team can optimize those weaknesses in the service. Usually, some big faults in the server configuration or server side scripting are discovered at this phase. In our experience, basic properties such as database query implementations and even chosen image formats can cause serious performance issues depending on choices made in design and development. Besides basic verification of the basic functionality of the SUT, our goal was also to verify the performance of the system and identify any bottlenecks.

We view test automation as a form of a pyramid, where manual testing is at the bottom and MBT at the top. This is illustrated in Figure 1. In this pyramid, the different levels of test automation build on top of the lower levels, and in our experience, it is not possible to implement the higher levels effectively without first having the lower levels working. This is similar to, for example, how Blackburn et al. described evolution of test automation from basic test execution to scripting based on action words, and finally model-based testing at the fourth generation [2].

Manual test automation at the lowest level can be just a user clicking on controls of a graphical user-interface and observing how they feel it should behave. Test scripts are typically a form of computer program written to perform a specific sequence. Keywords are abstractions that are transformed into test scripts by a test automation framework, allowing one to create test cases using higher level language concepts. As MBT generates test cases, optimally it should be able to generate them in terms of higher level elements such as keywords to avoid having to put low level details in to the test model. Keywords are a suitable approach for this as they are already supported by several test automation frameworks. An example of such integration can be found, for example, in [3]. An approach where the MBT tool can also be guided through embedded keywords, effectively combining benefits of both approaches can be found in [4].

In terms of a web service such as the one we tested here, this type of an effort is not directly possible, but rather scripting tools are required for even the most basic testing, where the reference for expected behaviour is typically the natural language specification of the SUT. The ability to execute test scripts is also a prerequisite for MBT. For this reason, the MBT part also requires having the same underlying execution platforms available as the manual scripting part we described as the first approach applied in our case study. The MBT approach also requires formalizing the specification as a suitable behavioural model from the typical natural language form. For this reason, we can only expect to have to spend more effort on the MBT approach. Thus, it is also important to understand the potential benefits to be able to evaluate where it may be most applicable and produce realistic gains.

Besides the choice of the level of test automation applied, other factors also affect the overall cost of the solution. This includes integration with other tools in the tool chain such as test management tools, defect tracking, continuous integration, virtual machines and others (see e.g., [5], [6] for examples). However, these are common requirements for any level of test automation and as such we focus here on the parts specific to test automation itself, where the differences are greater.

A. Previous Experiences

Test automation has been considered one of the biggest cost factors in the software development process for a long time. For example, Persson and Yilmazturk describe establishing test automation as a high risk and high investment project in their experience report [7]. They also list 32 pitfalls encountered in taking test automation into use in practice. These are too numerous to list all here, but some of the most relevant ones include poor decision making with regards to what is automated and to what extent, considering full test automation as a replacement for manual testing, and the misconception that test automation would always lead to savings in labour costs. We provide in this paper some added insights into these pitfalls and information to help make more informed decisions on what, where, and how to automate.

A similar experience report is also provided by Berner et al. [8]. Among other things, their experience shows misplaced expectations for fast return of investment of test automation, limited test automation beyond test execution, wrong abstraction levels for tests. They also note again that it is also their experience that automated testing cannot replace manual testing, but should rather be seen to complement it. Similarly, importance of proper maintenance is also emphasized. Again, we provide in our case study information on our experiences in manual vs. highly automated testing

 ISBN: 978-1-61208-233-2

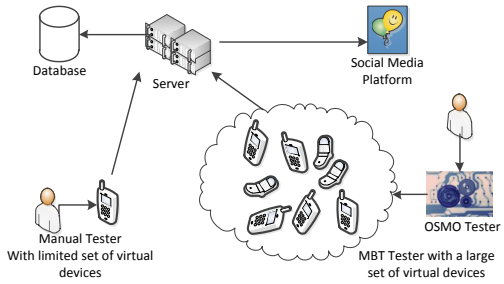

Figure 2. Description of the target service and the testing methods.

approaches (from the lower levels of the pyramid to the higher levels).

Several case studies on using MBT have also been published. These typically focus on presenting the benefits achieved, while providing little discussion or comparison with a manual test automation approach for the same SUT. Examples of such studies include MBT for the Android smartphone platform [9], for the healthcare domain [10], automotive domain [11], information systems [12], and several others. In this paper, we describe not only the benefits of MBT as a separate study, but a comparison with manual testing for the same project. Both testing methods aimed for the same goal, but ended up using a slightly different approach.

B. Our Testing Domain

The SUT here is a multi-user service with mobile phone clients and a backend server. A diagram of the environment is presented in Figure 2. The backend server also hosts web applications for social media purposes. The mobile client is a standalone application that collects information about the user behaviour and provides a view for the user to this information on the mobile device. The user can also activate a social media component, in which case the data is uploaded to the backend server once a day. When activated, the backend server processes the data and provides a summary of information through a social media application.

Data collected by the mobile phone application consists of logged events with timestamps and usage information of different applications on the phone. The data is XML formatted, but also contains comma separated data fields within.

The backend service receives the data sent by the mobile clients and stores it in temporary data storage. After a certain time, it is batch processed to a more detailed form. It is then stored in the final location to be used by the social media application. The triggering of the batch process causes high spikes in the service load. Due to the heterogeneous user set, the social media also causes high load in the form of complex database queries with varying parameters based on different user configurations and their associated social network graph.

The desired capacity of the service was calculated to be around 100 000 data uploads per day and the same amount of social media content requests per day. The registration message, which is only sent once per user, is not included in the estimates. Numbers were calculated from the estimated user count of 1 million from which one tenth were expected to enable data upload to the backend server, and little less were expected to also use the social media application. The service was expected to have users globally so the load is distributed evenly around the day. Little higher service load was expected to occur in the European daytime as the biggest set of users were expected to be from Europe. Size of the data was estimated to be 4 KB for every data upload. These numbers were safe estimates to leave some room for error. This would simply calculate into 400 megabytes of data each day and roughly 1 upload per second.

The estimates are hard to make for such a service with different users and mobile phone capabilities. Also, the popularity of the social media application varies between countries and its usage is hard to estimate. Therefore, the distribution of request types sets the problem for estimating the capabilities of the service and therefore the testing of the service. As it is a web service it faces the same problems as any other service with multiple users. An estimate has to be made as to what kind of requests can be expected in what ratio and in what sequence. If the more data intensive requests are more common, then the required capabilities of the service are different than in a situation when it mostly faces computing load and only little bandwidth load.

To create test scenarios to test this type of service, a tester needs to write complicated test scripts. For example, a tester could write scripts that create requests to the server and have the script repeat these requests any number of times. This script could also include a mechanism to observe if the system responded properly and if the data gets processed into final storage properly. Checking the data from the storage requires a secondary access point to the database. Last step would be to make sure everything gets done properly no matter what the sequence of actions is.

The input in testing was chosen to contain the most sensitive and data intensive requests types. We will call them request1 and request2. Request1 contained user registration information with many details about the user and the mobile device. Request2 contained certain logged information collected during a day from the phone. This data was naturally linked to the previously created user.

III. Manual Testing

During the main development phase of the system, the manual test approach was the one first applied to create a limited set of required test scenarios. Here, by manual, we mean simple scripting based testing. It was seen as a means to quickly achieve the needed coverage for the most critical requirements.

 ISBN: 978-1-61208-233-2

Apache Bench[13], Siege[14], Grinder[15] and Curl[16] were used as tools for the testing in this phase. The goal was to verify the core functionality of the service, and to see how many requests it can serve in a certain time.

A. Process

1) Create desired requests: Using Curl, a simple HTTP request program, the tester first created a request and verified that it gets appropriate response. Variables in the request were static.

2) Implement the request into test program: When the request type is clear and the parameters are correct, this request can be implemented as a test case for Apache Bench, Siege or Grinder. Both were used in this test and they provided similar features for this type of simple testing. Apache Bench cannot be used as versatilely as Grinder. Grinder provides the possibility to define test case with multiple requests which is useful in this scenario. Apache Bench was only used to test performance with one type of request.

3) Scripting: Apache Bench runs a single HTTP request with defined amount of threads for defined amount of times. To run repeatedly with different parameters in repeats and concurrence the tester has to implement a small shell script to make this process easier for repetition purposes too. The same applies for Siege and they are both strictly command line operated.

4) Execute the tests: Executing here means letting the test script repeat the test with desired configuration. In Siege and apache bench the testing can be configured by changing the amount of simultaneous requests and interval in requests. The goal was to reach a certain performance in terms of requests per second with any reasonable count of simultaneous requests. In practice, we targeted 50 to 200 simultaneous requests. Simultaneous users or requests here mean executing the HTTP requests as fast as possible with parallel threads. The same number of simultaneous users in a web page consumes fewer resources than this because users do not repeat their requests that fast.

5) Observe performance: Documenting and analysing the results is the final part of testing. Server performance was analysed with one type of request at first. This way, some sorts of estimates of the performance were determined and some rough limitations could be set. In reality, other variables and requests can have great impact on performance.

B. Weaknesses

During the initial testing phase many weaknesses were noticed in the manual testing process. The test scripts were not able to produce variance in the test data. This was partially solved by creating random variables in the test data. However, it still did not produce suitable data when certain type of data was needed in different scenarios. When the test script is not designed from the beginning to be functional enough with variables and randomness, its configurability is weak.

IV. Model-based testing setup

MBT is often used to help testers increase test coverage. This can be achieved by varying the test sequences or variables in test steps through the usage of the model. In our test environment we have used the OSMO Tester MBT tool described in [17].

MBT and test automation in general are processes that require time for setting up and implementing. Their advantage is usually observed in a longer running software development project.

In our case, we used only MBT with the goal of fully integrating all the required external testing tools into the MBT tool. This way, the test generator could directly generate and execute test steps against the SUT. This way, the test engineer does not need to set up complex execution environments for different tests, but can run them at once. The tools used were Grinder, OSMO and JDBC SQL connector. To integrate this environment the solution was to import OSMO as a library to Grinder Jython based environment and use the test generator from there. Similarly it was also possible to use JDBC connector for the database connection.

A. Process

1) Grinder: The first phase of building the MBT setup was setting up Grinder with the requests and response verification. The main goal was to see how Grinder works and how it could be integrated with OSMO to perform the testing against SUT. Here, the input data, along with user and request count configuration, was specified inside Grinder.

2) Grinder and OSMO: In the second part of the process, the OSMO was brought into the setup. The possible requests and responses were included in the OSMO system test model. This model was used in Grinder Jython script which then produced input data for the Grinder HTTP requests. This way, the requests had increased coverage while still being able to verify the responses.

3) Grinder, OSMO and SQL: The last part of the MBT setup was bringing the database element into testing. Naturally to verify the server operation the database processors correct operation had to be confirmed. Bringing this into the test automation really improved the testing process in our case. In manual testing, verification was limited to checking one or few requests ending up correctly into the database after processing. In the MBT setup, the effect of every request was checked against the database.

B. Weaknesses

As mentioned many times in reports about model-based testing, the launching and covering the requirements takes lot of time in the beginning. For us, also the matter of learning the environrment delayed the process of implementing model-based testing. A decision about where to

 ISBN: 978-1-61208-233-2

implement the model had to made. We decided to create a Java class including the model which was then brought to the Jython based environment of Grinder. This setting up phase also lead us to broaden the requirements that we would cover with the setup. It felt the limited set of testing requirements set by the project was not worth implementing as a model-based setup. Therefore, we went on to implement a more advanced performance testing setup which included the validation of performance in malicious and exceptional situations.

V. Discussion

The manual testing in the project was done by a test automation expert who had a long history with the project and its previous iteration. He also had experience with web applications and the tools used for manual testing prior to the manual test effort. He was able to get going and implement the first test cases in a matter of hours. Further test cases were implemented over time and were similarly low effort. However, they were limited to testing only the core features with as few test scripts as possible. The manual testing reached higher coverage in terms of covering all the features of the service. Despite this it lacked in covering those requests with different types of parameters.

The MBT testing in the project was done by a test automation expert who had his background in a different domain, was unfamiliar with the SUT, and with the tools used to implement model-based testing for the SUT. Learning the tools and the expected behaviour of the SUT took several months of effort to build the different iterations of the MBT solution described in Section IV. Each iteration took about one man-month of effort in this case. The MBT implementation of the testing covered the request types in the web service that are most resource intensive. The main goal was to provide more variance in the performance testing which was the main concern in deploying of the service.

A. Method evaluation

Evaluating the difference between processes is difficult in a case such as this when the two in comparison have different components to start up with. The person performing the manual testing had different background than the person building the model-based testing. Therefore, the efficiency and performance of the processes are hard to evaluate, but we concentrate on the performance and usability of the test environments which came out as a result.

1) Setting up: As stated earlier the difference in setting up these two types of test environments is noticeable. Model-based testing takes more time in early stages, but saves time in longer run. This is often shown in research about model-based testing[18]. This effect was clearly visible, especially in the great effort of setting up the model-based environment in contrast to ease of making the manual tests run accordingly.

2) Repeatability: Repeating the created set of requests was not an issue in manual testing. When a set of tests and scripts were done, repeating those and changing the user and request count was easy. In this sense manual testing can easily execute the test cases as a form of regression testing. This is important, especially when something is changed in the SUT and there is need to test the performance for possible improvements. MBT can cause issues here if care is not taken to make the generated tests repeatable. If the test cases are regenerated from a changed model or from the same model with different parameters, the test contents can change and the results are not consistent. Because of this care has to be taken to document which model configuration or which set of generated tests was used for which reported performance.

3) Modification: At one point in manual testing there was a certain set of requests that repeated in many test cases, but in a different sequence. When a new request type was added it had to be added to every test case, which was very laborious. Making this type of change in the MBT setup does not require this type of effort. The tester only needs to add the request type into the model and generate new tests with varying sequences and payloads.

4) Sequence direction: In manual testing, the tester has to create the sequence by hand and this repeats through the whole test run. This is true if Siege is used to define the test case requests. In the MBT approach we used sequence direction for example by defining that request1 has to repeat n times before any request2 type requests are generated. Support for rules like these makes it easy to produce guided variance in test generation.

5) Request type distribution: This part was the most difficult for manual testing with the tools we used. Siege was the tool that was able to take the requests as a list and therefore with some added manual effort it was possible to modify the ratio of request1 and request2. Still this was not that flexible. OSMO provides the possibility to simply give a weight to each step in the test case and this way the ratio of each request can be defined to a great accuracy in a long test case. Different variant combinations can also be easily generated to any numbers automatically with OSMO Tester.

6) Payload configuration: The biggest advantage of MBT in this scenario was the ease of modifying the content of the requests. This was illustrated earlier in Figure 2. It was possible to make each request have different type of input data and to include some faulty data. Even though this might not be essential in performance testing, it is important to know whether certain type of data causes performance issues or lockups.

VI. Conclusion

We have described here a set of experiences in implementing both manual and model-based performance testing for a single project. The results show how manual testing was an

 ISBN: 978-1-61208-233-2

effective way to bootstrap the testing process and produce focused test cases for the system under test. With good knowledge of a large set of suitable tools for the domain, it is also possible to create a good initial test suite with reasonable effort.

However, maintenance of large test suites quickly becomes laborous, and addressing extensive variation in testing manually is too expensive. This is where MBT can help. MBT can have a significant initial investment required, but can result in much easier means to evolve a test suite and to address large scale variation requirements. Familiarity with the tools, techniques and the domain can also work to significantly reduce the initial investment required.

In the end, we can say that for us the best process would be to start with manual testing and when the system under test and the test environment are stable enough bring in model-based testing. At this point, MBT can also be applied to address some of the needs for manual testing using specific configuration properties of OSMO Tester such as those described in [4].

We continue to apply MBT to new projects with similar and different properties, collect the experiences and improve our understanding of how the different types of testing may benefit the testing process in different contexts. We wish to have a case to properly apply and analyze both manual and model-based approach. In this case the model-based approach was added later then whe possibility appeared. This way, the starting points and motives of the testing were not exactly same. With a dedicated case we could provide a better comparison of the types with metrics and cost analysis. In the beginning of each process the testing had same goal. However when model-based testing advanced, its purpose was altered a little to cover slightly different objective.

REFERENCES

[1] A. Shahrokni and R. Feldt, "Robustest : Towards a framework for automated testing of robustness in software," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 78–83.

[2] M. Blackburn, R. Busser, and A. Nauman, "Why model-based test automation is different and what you should know to get started," in *International Conference on Practical Software Quality and Testing*, 2004, pp. 212–232.

[3] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," in *4th IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 242–251.

[4] T. Kanstén and O.-P. Puolitaival, "Using built-in domain-specific modeling support to guide model-based test generation," in *7th Workshop on Model-Based Testing (MBT 2011)*, 2012.

[5] B. Peischl, R. Ramler, T. Ziebermayr, S. Mohacsi, and C. Preschern, "Requirements and solutions for tool integration in software test automation," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 71–77.

[6] V. Safronau and V. Turlo, "Dealing with challenges of automating test execution architecture proposal for automated testing control system based on integration of testing tools," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 14–20.

[7] C. Persson and N. Yilmazturk, "Establishment of automated regression testing at abb: Industrial experience report on avoiding the pitfalls.," in *19th International Conference on Automated Software Engineering (ASE'04)*, 2004, pp. 112–121.

[8] S. Berner, R. Weber, and R. Keller, "Observations and lessons learned from automated testing," in *27th International Conference on Software Engineering (ICSE'05)*, 2005, pp. 571–579.

[9] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, 2011, pp. 377–386.

[10] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying model-based testing to healthcare products: Preliminary experiences," in *30th International Conference on Software Engineering, (ICSE 2008)*, 2008, pp. 392–401.

[11] E. Bringman and A. Krmer, "Model-based testing of automotive systems," in *3rd International Conference on Software Testing, Verification, and Validation (ICST 2008)*, 2008, pp. 485–493.

[12] P. Santos-Neto, R. Resende, and C. Pádua, "An evaluation of a model-based testing method for method for information systems," in *ACM Symposium on Applied Computing*, 2008, pp. 770–776.

[13] "Apache bench, the apache software foundation," http://httpd.apache.org/docs/2.0/programs/ab.html, 2012, [retrieved: June-2012].

[14] "Siege load tester home page," http://www.joedog.org/siege-home/, 2012, [retrieved: June-2012].

[15] "The grinder, a java load testing framework," http://grinder.sourceforge.net/, 2012, [retrieved: June-2012].

[16] "cURL command line tool," http://curl.haxx.se, 2012, [retrieved: June-2012].

[17] T. Kanstén, O.-P. Puolitaival, and J. Perälä, "Modularization in model-based testing," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 6–13.

[18] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

 ISBN: 978-1-61208-233-2