Data-driven test automation: augmenting GUI testing in a web application

by

Erik Kurin, Adam Melin

LIU-IDA/LITH-EX-A--13/043--SE

2013-07-10
Final Thesis

Data-driven test automation: augmenting GUI testing in a web application

by

Erik Kurin, Adam Melin

LIU-IDA/LITH-EX-A--13/043--SE

2013-07-10

Supervisors: Pär Emanuelsson (LiU), Kristian Karl (Spotify)
Examiner: Prof. Kristian Sandahl
Abstract

For many companies today, it is highly valuable to collect and analyse data in order to support decision making and functions of various sorts. However, this kind of data-driven approach is seldomly applied to software testing and there is often a lack of verification that the testing performed is relevant to how the system under test is used. Therefore, the aim of this thesis is to investigate the possibility of introducing a data-driven approach to test automation by extracting user behaviour data and curating it to form input for testing.

A prestudy was initially conducted in order to collect and assess different data sources for augmenting the testing. After suitable data sources were identified, the required data, including data about user activity in the system, was extracted. This data was then processed and three prototypes were built on top of this data. The first prototype augments the model-based testing by automatically creating models of the most common user behaviour by utilising data mining algorithms. The second prototype tests the most frequent occurring client actions. The last prototype visualises which features of the system are not covered by automated regression testing.

The data extracted and analysed in this thesis facilitates the understanding of the behaviour of the users in the system under test. The three prototypes implemented with this data as their foundation can be used to assist other testing methods by visualising test coverage and executing regression tests.
Glossary

API - Application programming interface. A defined specification for an application.

CSS - Cascading Style Sheets. Used for styling elements written in a markup language such as HTML.


Data warehouse - A central repository for data, used for reporting and data analysis.

GraphML - XML-based file format for storing graphs.

GraphWalker - A tool used for MBT. More information at www.graphwalker.org.

GUI - Graphical user interface. An graphical interface which the user can interact with the software.

HTML - Hypertext Markup Language. A markup language used for creating web pages.

iFrame - An inline frame in a HTML document.

JavaScript - Programming language often used for interacting with web sites.

JSON - JavaScript Object Notation. Open standard for representing simple data structures for data interchange.


MBT - Model-based testing. A software testing approach.

Python - Programming language which supports multiple programming paradigms.

RDBMS - Relational database management system. Database management system based on a relational model. E.g. MySQL and Oracle database.

SUT - System under test. The system which is being tested.


Zettabyte - 1 000 000 000 terabytes.
Acknowledgement

We would like to give our sincere thanks to all the friendly and helping people at Spotify that made us feel part of the family and helped us along the way. Specifically, we would like to thank our supervisor Kristian Karl who we learned many important lessons from regarding software testing and who helped to make a complex subject easy to understand.

We would also like to extend our deepest thanks to Kristian Sandahl and Pär Emanuelsson of the Department of Computer and Information Science at the Institute of Technology at Linköping University for their guidance and invaluable advice during the work of this thesis.
Contents

1 Introduction 1
  1.1 Background .................................................. 1
  1.2 Spotify ......................................................... 1
  1.3 Problem Description ........................................... 2
  1.4 Goal ............................................................. 2
    1.4.1 Research questions ......................................... 3
  1.5 Limitations and scope .......................................... 4
  1.6 Disposition ..................................................... 4

2 Theoretical framework 5
  2.1 Model-based testing ............................................ 5
  2.2 Distributed computing ........................................ 7
    2.2.1 MapReduce .................................................. 7
    2.2.2 Hadoop ...................................................... 8
  2.3 Data mining and probability theory 8
    2.3.1 Markov chains .............................................. 9
    2.3.2 Cluster analysis .......................................... 9

3 Technical background 11
  3.1 Testing at Spotify ............................................. 11
    3.1.1 Model-based testing ...................................... 11
  3.2 Data analytics at Spotify ..................................... 13
    3.2.1 Data collection .......................................... 13
    3.2.2 Data analysis ............................................ 14

4 Related work 15

5 Method 16
  5.1 Pre-study ..................................................... 16
  5.2 Data collection ................................................ 17
  5.3 Implementation approach .................................... 18
  5.4 Development process ........................................ 19

6 Results 20
  6.1 Data collection ............................................... 20
    6.1.1 Web player events ....................................... 21
    6.1.2 Test user statistics ..................................... 21
  6.2 Data visualisation ............................................ 22
    6.2.1 Plotting .................................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2 Overlay</td>
<td>22</td>
</tr>
<tr>
<td>6.3 Test Runner</td>
<td>24</td>
</tr>
<tr>
<td>6.4 Model generation</td>
<td>25</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>28</td>
</tr>
<tr>
<td>7.1 Future work</td>
<td>30</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>32</td>
</tr>
<tr>
<td>8.1 Implementation status</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The problem description visualised ................................................. 2
1.2 How the presented problem is proposed to be solved by extracting data about users ................................................................. 3

2.1 An example of MapReduce operations on a data set ......................... 8
2.2 An example of a simple Markov chain ........................................... 9

3.1 A graph representing the login flow of the Spotify client .................... 12
3.2 The Spotify backend architecture .................................................. 13

5.1 Flow chart visualising the work progress of the thesis ........................ 18

6.1 Overview of the MapReduce jobs and their applications ..................... 20
6.2 Bar chart representing the differences between regular users and test users ................................................................. 22
6.3 Architectural overview of the overlay visualisation ............................ 23
6.4 How output of the MapReduce job TestUserStatistics is converted into CSS properties ............................................... 23
6.5 A screenshot from the Spotify web player with color animated elements which highlights weak test coverage ........................................ 24
6.6 How an event is converted to a test method ...................................... 24
6.7 Architectural overview of the Test Runner ...................................... 25
6.8 The SSE and silhouette values for different k values .......................... 26
6.9 An overview of the Model Generation solution ................................ 27
List of Tables

2.1 Transition matrix for the markov chain in figure 2.2. . . . . . . . . . . . . . 9
6.1 An example of the output from the WebPlayerEvents MapReduce job. . . . 21
Chapter 1

Introduction

This master thesis was carried out during the first half of 2013 at Spotify’s headquarters in Stockholm.

1.1 Background

The art of software testing has existed for almost as long as software development and has undergone several different fundamental shifts. The objective of software testing however has remained the same for decades, to evaluate the quality of the system under test (SUT) and identify areas for improvement by finding defects and problems[4]. One method that has emerged and gained popularity during the last decade is model-based testing (MBT), appreciated for its ability to provide an abstraction layer between the design and the implementation of the tests.

The number of connected users and devices to the internet has exploded during the last decade and more than a third of the world population is using the internet[17]. This fact together with the ever increasing storage capacity and availability of the internet have popularised the term 'Big Data’. This term does not define the size of the data set or its application but rather functions as an umbrella term for data sets which conventional databases cannot store and analyse efficiently due to the size. Current predictions for all digital data created and replicated during 2013 are estimated to reach 4 Zetabytes (up 50% from 2012) and that spending on Big Data technologies will reach $10 billion during the same time period[6]. Rapid advances in technology spearheaded by data-driven companies such as Google, Facebook and Amazon have made it possible to process Big Data efficiently and inexpensively for both enterprises and consumers.

1.2 Spotify

Spotify is a Swedish-founded company which provides a service for streaming on-demand music. Launched in 2008, the service now has over 20 million active users in 17 different countries. 800 employees work across offices in Sweden, the United Kingdom, the United States and in several minor offices throughout the world. The work of this master thesis is carried out in the Technology department of Spotify and more specifically within the Test Automation (TA) and Analytic teams.
1.3 Problem Description

Traditionally, software development has been carried out in a sequential process, for example according to the waterfall model. In this context, testing is assigned a long phase at the end of the process to ensure quality of the system before release. Today, however, the ability to quickly release new features and shorten the feedback loop is increasingly important for many organisations. Agile and lean development methodologies with short development cycles are thus gaining popularity. These development methodologies require the testing to have a more integrated role in the entire development cycle and not just a phase at the end of the cycle.

One approach to integrate testing into an agile development process is to practice MBT. The rapid iteration speed of agile processes results in the need for regression testing and a testing method which is maintainable, which MBT provides. MBT also provides an abstraction layer between the design and the implementation of the tests. The models that the actual testing is based on are usually created manually and then parsed to automatically generate test cases. A problem shared with other testing methodologies is that the eventual test execution might not coincide with the behaviour of the real users of the SUT. Thus, in order to augment the testing at Spotify, there is a demand for a more data-driven testing approach and techniques for verifying that the testing is relevant in this sense as illustrated in figure 1.1.

![Diagram](image)

*Figure 1.1: The problem description visualised.*

1.4 Goal

Spotify possesses a huge amount of data and this is one of their greatest assets. Examples of data that Spotify handles is logging of user interactions and statistics of streaming and
performance. The data is used widely across the organisation, e.g. to pay stakeholders and continuously improve the service and it is an endeavor for the company as a whole to become increasingly data-driven. The automated testing, however, is still solely based on models of the SUT that are created manually.

Therefore, this thesis aims to investigate the possibility of making the test automation more data-driven by extracting user behaviour data and curating it to form input for testing as can be seen in figure 1.2. Part of the goal is also to develop simple frameworks that demonstrate these concepts and that testers can use. Another objective of the thesis is to provide verification of the testing relevance with regard to the user behaviour in the system.

![Figure 1.2: How the presented problem is proposed to be solved by extracting data about users.](image_url)

1.4.1 Research questions

The following questions are guiding the work of this thesis:

**RQ1:** Is the current testing at Spotify relevant with regard to the behaviour of real users in the system? If not, how can the testing process be improved in this sense?

**RQ2:** How can common patterns in the behaviour of the SUT users be modeled?

**RQ3:** How can user behaviour data, such as event logging messages, form input for test automation?
1.5 Limitations and scope

There are many kinds of testing techniques on different levels. To limit the scope of this thesis, it focuses on automated GUI testing on a system level, so no other types of testing will be discussed.

Another limitation made was to study only one of the numerous Spotify client softwares that are available for different platforms. The main reason for this was that the logging format was not standardised and differs a lot between different clients, which complicates the processing of this data. The client chosen to focus on was the Spotify web player, as it had the most extensive and uniform logging that allowed for extraction of user behaviour data.

1.6 Disposition

This report begins by introducing the reader to the necessary theory in chapter two. Chapter three describes the technical context in which the thesis is conducted. In chapter four, related work is presented. Then, chapter five explains the methods that are employed throughout the thesis. In chapter six, the results are presented, followed by a discussion in chapter seven. Lastly, chapter eight concludes the report with some conclusions.
Chapter 2

Theoretical framework

In this chapter, the fundamental theoretical topics are introduced for the benefit of the reader. An overview of the evolution of software testing and the concepts of MBT are initially introduced. Distributed computing with focus on MapReduce and Hadoop follows and the chapter concludes with the theory behind the data mining concepts which are utilised in solutions presented in chapter 6.

2.1 Model-based testing

When dealing with a complex subject, a common approach to help with understanding is to make a model of it. A model represents the characteristics and properties of the subject at a level of detail which is decided by the author. Another way to define a model is as a simplification of a subject in order to better understand it.

The idea behind MBT is to describe the expected behaviour of the SUT by creating an abstract model[5]. This model should contain enough detail to describe the SUT behaviour sufficiently and small enough to be produced cheaply[18]. A test automation tool then takes this model as input and produces abstract test cases from the model. The abstract test cases can be concretised in the language or testing framework of choice. The execution of the tests can be monitored and controlled by a test execution tool which can use the previously defined model to augment the testing by using the model’s properties and boundary conditions. A concrete example of this can be found section 3.1.1.

To better understand the concept of MBT, it is important to know how it has evolved from earlier testing methodologies. Manual testing is the earliest style of testing where a test plan is created from the requirements. The test plan defines which functionality to test and the scope of testing among other high-level objectives. Designing the test is also done manually as well as executing the tests and reporting the test results. This results in re-executing of the tests becomes expensive[12].

Capture/Replay testing is developed to reduce the cost of re-execution by recording the manual testing actions and replay them when needed. A major drawback of this process is the sensitivity to changes in the SUT[9]. If one minor detail is changed in the SUT the test has to be re-executed and captured again.

To further reduce the cost of re-execution, a script-based process is developed with the
goal to fully automate the test execution. Scripts are created to set up the SUT to a
desired state, passing input values and assign pass or fail to the tests depending on the
output. This approach adds two constraints on the organisation[18]. Testing the SUT
could previously be done by anyone that could interact with it but with the scripting
approach the tester must be able to develop these scripts, using a standard programming
language or a domain-specific testing language. Secondly, in order for the scripts to interact
with the SUT a predefined interface needs to be established, this demands a SUT that has
testability built in. Since testability is viewed as a quality criteria[8], this should not
be much of a problem, however it can prove expensive to add testability to an existing
product compared to doing it from the start. It is not uncommon that the size of the test
scripts rivals that of the SUT which depending on the test script developer could result
in a maintainability problem. Each change in the SUT or in the requirements must be
reflected in the test scripts.

To overcome the maintainability problem of the scripting approach the level of abstraction
needs to be raised[18]. The keyword-driven automated testing process adds a layer above
the test scripts called keyword/action-word framework while parameterising the test scripts
and making them as generic as possible. The idea is to combine keywords and action-words
and translate them to executable test scripts by using an adapter. For example, a table can
be constructed with each cell containing a keyword which maps to a test implementation.
Each row in this table corresponds to a test case and by inputting new rows into this table
new test cases are created. By raising the abstraction level the test design can now be
done by a non-programmer[10].

Each of the approaches to testing mentioned above tries to ease the cost of execution and
re-execution. However, there are still some unsolved problems present[18]. The cost of
maintainability can still be reduced and the design of test cases is done manually which
keeps the costs up. Tracing the coverage of requirements from the test cases is a process
which is done manually as well. With this as the basis, MBT raises the abstraction further
and the automation of test design and requirement traceability.

The workflow when doing MBT begins with the test designer creating a model of the expected
behaviour of the SUT and inputs it to the test case generator along with parameters
defining the coverage criteria. The criteria is different depending on how you choose to repre-
sent the model but it could be what minimum percentage of the model should be covered
or a stop condition, among others. Abstract test cases are output and depending on the
requirements of the SUT and the features of the test generator tool a coverage report can
also be outputted. The coverage report outlines how much of the model that is covered.

Since the model in most cases can be covered by an unlimited amount of different paths, a
large quantity of test cases can be automatically designed and generated[18]. The abstract
test cases are then converted to executable test cases either automatically by templates
or manually by a test programmer. The execution of the tests can either be managed by
the model-based testing tool, then called online model-based testing or in an offline mode
where the concrete tests are executed against the SUT by an existing testing tool.

Utting et.al[18] lists the benefits of MBT as the following. The separation of the design
from the test implementation which enables the test designer and test developer to work
concurrently while also raising maintainability. Improvement of test quality since the
process of automatically creating tests from a model is systematic and repeatable. This in
turn means that the quality of tests can be measured over time given their coverage of the
model.
In closing, MBT can be summarised as the automation of the design of black-box testing[18].

2.2 Distributed computing

As the number of connected devices to the internet continues to grow, more and more data is generated. The capacity to store the data is growing meanwhile the price of doing so is decreasing. These two factors combined have contributed to the possibility of making decisions based on large amounts of data. However, in order to take advantage of the possibilities the analysis of the data needs to be efficient and fast. While the computing performance continues to improve year after year the amount of data does as well, to the extent that it is not feasible to store and analyse it using traditional databases and workstations. Another solution is needed and distributed computing has several advantages that ease process of large data analysis.

Distributed computing in itself is not a new phenomenon, using distributed systems for computational tasks have existed for decades. It is a cost efficient solution since you can use commodity hardware for the clusters of computers instead of one expensive supercomputer. As there are several machines involved, there is also an increase in fault tolerance, since there is no single point of failure. Scaling both up and down is also easier compared to a monolithic system. These properties of distributed computing is a good fit for data analysis and during the last decade the MapReduce programming model have gained a large following. The following sections describe the theory behind distributed data analysis and implementations of it.

2.2.1 MapReduce

Originally presented by Google in a 2004 paper[3], MapReduce is a programming model for processing and generating large data sets. It is developed with the aim of offering a solution which would lower the complexity of data handling, parallelism and error handling.

The basis of MapReduce is made up of a Map-method and a Reduce-method. Inspiration for the methods originate from the functional programming paradigm where the map method applies a function to a record and yields a key-value pair. The reducer method then applies a function to the records sharing the same key and outputs the result. A basic example showing how MapReduce works can be found in figure 2.1. The data set is split and distributed across n mapper computer nodes. These nodes can be in the same machine cluster or spread out across several. In figure 2.1 each mapper yields a key-value pair from each record in the data set. Since there are only two different keys k1 and k2 only two reducer nodes are needed. In each reducer all values corresponding to one key are collected and further logic can be applied to the subset. In the example all records containing the value aa are outputted.

Computations can be parallelised with ease since the data set is broken down into smaller subsets. This provides a scalable solution when the data set size changes since more computation nodes can be added or removed.
2.2.2 Hadoop

The MapReduce framework presented by Google and the accompanied filesystem Google File System[7] are proprietary and are not available to the public. However the notion of MapReduce spawned the open source Apache Hadoop project which implements the ideas presented in the Google papers. Hadoop have gained a large following of high profile companies since its inception at Yahoo, among those are Facebook, Microsoft and Twitter[13]. The Hadoop project have over time evolved and now contains several modules. For this thesis the MapReduce framework and the Hadoop distributed file system (HDFS) is of interest.

2.3 Data mining and probability theory

Today, the amount of data stored in the world’s databases is growing in a staggering pace. This evolution is increasing the importance of data mining as a means for describing structural patterns in this data. There are many techniques within this field and the ones that apply to this thesis are introduced in this section.

One problem that this thesis deals with is to model sequences of user actions in the Spotify client. Within probability theory, a possible approach to solve this is to use a stochastic process, where the state of the system is represented by a set of random variables. This implies that the state can have not only one but several values, each with a certain probability. The Markov chain, described below, is a discrete-time stochastic process, which means that the time domain of the process is discontinuous.

Another problem is to identify similarities between the behaviour of different users. In
this case there is no training set available with labeled data to provide a correct answer in advance. Thus, the supervised learning approach to machine learning does not apply. Instead, the problem is to find patterns in unlabeled data. This is referred to as unsupervised learning, where cluster analysis, described in section 2.3.2, is a common approach.

### 2.3.1 Markov chains

A Markov chain consists of a set of possible states and transition probabilities that links the states together. This chain has the Markov property, meaning that the next state only depends on the current state[11]. In other words the process has no memory of previous states. For every state there is a transition probability stating the likelihood of going to each of the other states. These probabilities can be either constant or functions of other variables and are usually expressed as a matrix. Figure 2.2 depicts an example of a Markov chain in the form of a directed graph, where the nodes A, B and C represent states and the edges are the transition probabilities between them. The corresponding transition matrix is shown in table 2.1.

![Markov Chain Diagram](image)

**Figure 2.2: An example of a simple Markov chain.**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0.65</td>
<td>0.35</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>C</td>
<td>0.7</td>
<td>0.3</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2.1: Transition matrix for the markov chain in figure 2.2.**

### 2.3.2 Cluster analysis

When it is desirable to divide a set of objects into natural groups, where objects in the same group are similar in some sense, cluster analysis is a suitable and commonly used approach. There are many clustering methods that attempt to solve this problem. The clusters that they create from input data represent similarities in the characteristics of the data that is taken into account when performing the clustering. These clusters can be either exclusive, so that an object belongs only to one cluster, or overlapping, meaning
that an object may belong to several clusters. Another approach is to have probabilistic clusters, where an object belongs to a cluster with a certain probability[20].

In this thesis, the cluster analysis approach used is the iterative k-means method. This method is named after the parameter k, that specifies how many clusters to divide the input data into. The k parameter is specified in advance, whereby k random positions in the input space are chosen as cluster centres. Then the objects in the data set are assigned to the closest cluster center. This step requires a distance measure to determine the distance between two data points. Normally, this distance is measured according to the ordinary Euclidean distance function[11]. When each of the data points have been assigned to a cluster, the centroid of each cluster is computed. These centroids represent the average of their clusters. The above procedure is then repeated with the centroids as the new central points of each cluster. This iteration continues until the clusters have been stabilised, which happens when the same points are chosen as centroids in consecutive rounds[20].

One main drawback of the k-means method is that the k parameter has to be specified before the clustering algorithm starts. This is a problem as it is often difficult to know in advance how many clusters there are in the input data. The result of the clustering might also be different depending on what points were chosen as initial cluster centres. These problems can be solved, although it is computationally expensive. By running the algorithm multiple times with different values for k and measure the quality of each clustering performed it is possible to determine a good value for k. Additionally, you can run the algorithm a number of times for each k with different points initially chosen as cluster centres, and then use the execution that gives the best result[11]. This approach requires metrics for measuring the quality of the cluster executions. A common such metric is the sum-of-squares error (SSE). The SSE value, computed according to equation 2.1, where \( y_i \) is the i:th data point and \( c \) is its cluster centroid, represents the sum-of-squares of the distances, usually Euclidean, between each data point and the centroid of the cluster it belongs to. Hence, a low SSE value implies that the data fits well into the generated clusters[15].

\[
\sum_{i=1}^{n}(y_i - c)^2 \tag{2.1}
\]

Another metric is the silhouette coefficient, computed for each data point according to equation 2.2, where \( a(i) \) is a measure of the dissimilarity between \( i \) and its own cluster, and \( b(i) \) measures the dissimilarity to the next nearest cluster. Then, an average silhouette is computed for all data points, and this value can be used to evaluate the clustering validity[14].

\[
s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{2.2}
\]
Chapter 3

Technical background

This chapter explains how the concepts introduced in the theoretical framework are used at Spotify. It also introduces terms and tools used throughout the Result chapter.

3.1 Testing at Spotify

A key strategy for Spotify is to be fast. An agile development process enables this speed but puts pressure on the quality assurance. When new features are developed rapidly the testing organisation has to prioritise between testing existing features or the newly created. It is desirable for the tester to focus on developing test for the new features in parallel with the developers in order to increase the quality. Test automation is the key to free up time for the testers so they can stay focused.

The regression testing of existing features is a repetitive, systematic and somewhat dull process for a human tester. A tester is more productive developing new tests where decisions and creative thinking is involved. In other words, the regression testing is suitable for automating which is done at Spotify. MBT enables the entire process from the inception of new features to their regression testing to go smoother.

3.1.1 Model-based testing

At Spotify, a tool called Graphwalker is mainly used for MBT. This tool is implemented in Java but there is also a re-implementation for Python available. Graphwalker generates test sequences from a predefined model. These test sequences are outputted as Java method stubs, in which the test developer implements the test for the SUT. The tests are then executed either by an existing framework or by Graphwalker in a so-called online mode. The following section describes the workflow when working with Graphwalker.

The models are finite state machines that are created manually using any software that can export to the GraphML file format. Each edge represents an action and the vertices are verification points which results in each action is associated with a verification. There can be several edges originating from the same vertex to different vertices, as well as recursive relationships with edges looping back to its former vertex. For an example of a model see figure 3.1 where the login functionality of the Spotify desktop client is tested.
In each model there is a start vertex but no end vertex. When generating test sequences an end criteria is specified, which is explained more thoroughly later in this section. Each edge and vertex is labeled with an identifier prepended with `e_` and `v_` respectively. As the test sequences are generated, the test method names corresponds to these labels. In addition to labels, conditional expressions, actions and keywords can be added to the edges while the vertices can be augmented with keywords further described below. In figure 3.1 there are three vertices present which verifies that the test client has reached a specific GUI screen. The edges are the available actions to perform to move from one screen to another.

The conditional expressions on the edges acts as guards. The expressions are evaluated by an internal interpreter and if they are evaluated to true Graphwalker proceeds to execute the edge. In figure 3.1, the edge `e_StartClient` can either proceed to vertex `v_LoginPrompted` or to `v_WhatsNew` depending on if there has been an valid login. Edges can also contain actions, which is a Java expression that is executed by the internal interpreter. This is useful to store information in variables which is accessible to the test developer inside the test cases or to the model. Continuing from the previous example, after the `v_LoginPrompted` vertex is executed, the edge `e_ValidPremiumCredentials` sets `validLogin` to true, indicating a successful login.

Both edges and vertices can have keywords added to them which extends the functionality of the model. The edges can be weighted with a percentage value in order to allow it to be executed more often and vertices can have requirement identification numbers added to them to mark a requirement as fulfilled when a certain vertex is reached. More keywords are available, for a complete listing please visit Graphwalkers webpage.

When the test designer have finished the model, Graphwalker takes the model as input and create java methods for the edges and vertices. The test developer then implements the code to test the SUT from these methods. Graphwalker also provides an API for communicating with the model during runtime from within the implemented test code.
In order to gain the additional benefits of MBT, Graphwalker handles test execution in a so-called online mode. When running in online mode, Graphwalker interacts with both the test code and the model. Previously it is mentioned that the models did not contain any exit vertices and the reason for this is because none are needed. When launching Graphwalker, along with specifying a model, a stop criteria is also passed as an argument. A stop criteria can be a certain percentage of edges/vertices covered for example. How the traversing of the model should be carried out can also be specified, either by a random walk or by the shortest path possible to fulfill the stop criteria.

The stop criteria creates a better coverage of the model and thus better SUT coverage as the path is varied. Another beneficial property of the stop criteria is that the same model can be used for different types of testing. When doing functional testing it might be enough to visit each edge/vertex pair once whereas when conducting stability testing it might be more suitable with a time duration as a stop criteria.

3.2 Data analytics at Spotify

As mentioned in the introduction chapter, Spotify is a data-driven company and the data helps business intelligence, recommendations and reporting among others. This thesis relies heavily upon the data infrastructure at Spotify for access to the data that constitute the foundation of the analysis and results. In the following section, a high level description of the infrastructure is presented.

3.2.1 Data collection

Communication between a user and the backend of Spotify is done through access points. Whether the user wants to authenticate or play a track, the access point handles the communication. Due to the access points’ role much of the data related to the users are collected here ranging from user interactions to network latencies. A simplified view of the architecture can be seen in figure 3.2. Since the size of the data collected consists of several Terabytes per day[16], HDFS is used for storage rather than a traditional RDBMS solution. The distributed nature of HDFS enables storage of large amounts of data which in turn can be processed efficiently by Hadoop.

![Figure 3.2: The Spotify backend architecture.](image)
3.2.2 Data analysis

Once the data resides on HDFS it can be queried and analysed using different solutions. The previously mentioned MapReduce is a powerful tool when analysing the data. Spotify has developed their own tool for scheduling MapReduce tasks called Luigi, which enables building complex pipelines of tasks. This includes dependency resolution, workflow visualisation among other features. Luigi is written in Python and as a consequence of this the MapReduce jobs are written in Python instead of Java which Hadoop is based on. The MapReduce jobs that are created consist of a mapper and a reducer method which handles the data, as described in section 2.2.1. MapReduce jobs can be chained together and run once or scheduled for periodical execution. The output of a job can then be imported into any program of choice for a deeper analysis. One drawback of the MapReduce jobs is the high entry barrier if the author does not have a software developer background, since the jobs are programs written in Python.

The data outputted from the MapReduce jobs can be queried using Spotify’s data warehouse or by MapReduce again by chaining jobs together. Spotify’s data warehouse service is built on top of Apache Hive. Apache Hive enables analysis of data stored on HDFS using a SQL-like language called HiveQL which during execution is converted into MapReduce jobs. This is advantageous since it leverages the power of the MapReduce framework while lowering the entry barrier by using SQL and a GUI. However, writing MapReduce jobs in Python offers a larger set of functions which makes the writing of complex jobs more manageable and flexible than using Hive.
Chapter 4

Related work

In this chapter, related work to the clustering approach and user analysis is presented. Analysing how users interact with a website is a topic often referred to as clickstream analysis, which shares the same characteristics as the problem of modelling common usage in this thesis. In order to categorise the users, clickstream analysis commonly uses a clustering method, that fall into either a model-based or similarity-based approach. For the model-based method, Markov models are frequently applied, while in the case of similarity-based methods distance and longest common subsequence are common choices.

Cadez et.al. present a model-based probabilistic framework for clustering individuals based on non-homogeneous data, such as observed navigational patterns[2]. In this example, a modified expectation-maximization (EM) algorithm is used to cluster the data. This approach is demonstrated on web session data in order to cluster users after their navigation history. The proposed solution tries to mitigate the two problems that when non-vector data is reduced to a vector, loss of information can occur and that when sequences have different length, the distance-based method have no way to account for this in the distance function.

Ympa et.al. propose a method which is an extension on Cadez et.al. and uses a mixture of hidden Markov models to automatically categorise and cluster users[21]. An EM algorithm is used to train the mixture models on artificial navigation patterns. Since the models learn the different page categories automatically no manual labeling needs to be performed on the data beforehand.

A solution using a vector space model called random indexing with weighted indexes is proposed by Wan et.al[19]. In contrast to the solutions above, the authors method aims to discover latent and unobservable factors in the user web browsing behaviour. The vectors are clustered by the k-means method and compared to other methods in a prefetching experiment.

Finally, Benerjee et.al.[1] introduce a method to categorise and cluster users using a weighted longest common subsequence. This method incorporates time information, which Markov-based methods are unable to do. The similarity between any given two navigational paths are calculated and mapped into a similarity graph, which is then used for clustering by using a graph-partitioning algorithm.
Chapter 5

Method

This chapter first depicts the pre-study that initiates the thesis work and specifies its scope. Subsequently, the approaches used for data collection and implementation are described, together with the development process methodology.

5.1 Pre-study

There were two major purposes of the pre-study. Primarily, the objectives and scope of the thesis had to be specified. Doing this required an understanding of the development and testing processes at Spotify, which was the second purpose of the pre-study.

To obtain an overview of how development and testing is managed at Spotify, a series of 13 interviews were conducted. Before the work of interviewing started, an interview plan was created to specify the desired output and how to achieve it. A large focus was put on the test automation approach at Spotify, since this is the main context of this thesis. Both the technical tools and the processes of testing were considered during the interviews. Furthermore, it was desired to understand how the test automation is integrated into the development process. Part of the plan was also to ask open-ended questions during the interviews in order to obtain detailed answers and retrieve as much information as possible.

To see the testing from multiple perspectives, the interviewees were chosen from a number of different teams and also from a number of different roles such as test automators, manual testers, agile coaches and managers. One of the purposes of the interviewing approach was to find out what the different roles think that the objectives and potential benefits are of having test automation.

During the interviews, both interviewers took turns asking questions and both also took separate notes of the answers. These notes were then compared and summarised to extract the most important outcomes of the interviews. The resulting summaries were sent out to each interviewee to make sure that they agree with the content and that nothing important had been left out.

In parallel with the interviews, a literature study was conducted to learn the basic principles of test automation and MBT. The initial selection of literature was selected in consultation with the thesis supervisor. Subsequently, this selection was expanded by performing searches in Google Scholar to find references to the most important works. To get an
overview of the literature, there were also regular meetings where each work was briefly summarised.

The output of the pre-study was a great help in organising the continued work on the thesis. From the interviews and literature study came knowledge about test automation and how it is practiced at Spotify. This knowledge is presented in the theoretical framework and the technical background. More importantly, the pre-study also resulted in a lot of ideas for the objectives of the thesis. Consequently, the scope was very wide at this point and thus had to be narrowed. When this procedure started, there were about 40 scope ideas. To reduce the number of ideas, they were all rated according to their suitability for an academic study and their potential benefits for Spotify. This way the number of ideas were reduced down to five. Then a discussion with the thesis supervisor began in order to determine the idea to use as a basis for the thesis. The resulting idea is described in sections 1.3 and 1.4. The other four ideas are briefly introduced below.

- Study the test automation process at Spotify and measure its efficiency and return on investment.
- Evaluate processes and tools for test automation on the Spotify Android client.
- Enhance the functionality by adding data analytics to the test result service that Spotify uses for reporting the results of automated tests.
- Investigate how test automation can be used to facilitate continuous integration.

5.2 Data collection

As the pre-study was finished, the thesis scope and goal had been specified, as seen in figure 1.2. Before starting to investigate the data-driven testing approaches, the actual data had to be produced. Due to the large amounts of data available, an investigation was required in order to determine what data to use and how to retrieve it. This investigation resulted in the decision to focus on a specific client event log message that is triggered by user activity in the client and then sent to the access points. It was estimated that this data is the most suitable for the purpose of this thesis, as it provides accurate information on user activity and, when processed, forms useful input for the testing. Furthermore it was decided to focus on a single platform, as there was a lack of uniformity in the client event logging between different platforms. Another reason for this decision was to make it easier to eventually integrate the developed techniques into the testing framework at Spotify. More specifically, the platform chosen to focus the thesis on was the Spotify web player. This application implements a recently developed framework for logging of client events. This results in a uniformity in the logging which makes the data a lot easier to process and eventually use as input for the testing.

Another decision regarding data collection that was taken was to use MapReduce jobs rather than Hive to access the data. The reason for this was that MapReduce enables access to more data sources. Besides, as the Luigi framework can be used for writing MapReduce jobs in Python, this approach has the advantage of using the full functionality of the Python language, which is more powerful than the HiveQL query language. This also makes it easy to maintain and collaborate on the code as the jobs are stored in plain Python files on the file system. To gather the data necessary for the analysis, two separate MapReduce jobs were developed. These are further described in section 6.1.
5.3 Implementation approach

The approach taken to the problem specified in 1.1 was first of all to access the data which to analyse according to the previous section. The next step was to figure out how to model user behaviour from this data. Concretely, this resulted in the production of GraphML files containing finite state machines that reflect common user behaviour and the use of these models as input to Graphwalker in order to generate method stubs according to the MBT approach applied at Spotify. To create these models, the output of a MapReduce job, containing a list of events, was parsed and the events were assembled into sessions. The sessions were then clustered with the k-means method and one session to represent each of the clusters was extracted and subsequently written to a GraphML file. This file format was chosen to make the models both parseable by Graphwalker and readable by the graph editor that is used at Spotify for creating models.

The stubs generated by Graphwalker from the GraphML files can then be implemented by calling test methods that are already implemented for other models. This solution, that is referred to as Model Generation from now on in the report, makes it possible to test the most occurring sequences of user actions.

The problem with the initial idea of the Model Generation approach, however, was that the abstraction level is lower for the event log messages than for the models usually created as input to Graphwalker. Therefore the orientation of this solution was slightly changed to instead parse a list of unique events and run tests of these individually. This new solution, referred to as Test Runner, is implemented in Java, in order to be easily integrated into the testing frameworks currently used at Spotify.

During the work on the above mentioned implementations, another implementation idea
emerged from discussions with the supervisor at Spotify. The idea was to extract statistics on how the activity of test users (accounts that are solely used for testing) differ from the activity of regular users. The behaviour of the two user groups are compared and the result is visualised. For this solution, that is referred to as Data Visualisation, there are two stand-alone implementations. First, a JavaScript implementation was developed to visualise the data as bar charts. The reason JavaScript was used, rather than simply creating the bar charts in Excel, was that this approach allowed for an automatic generation of the charts to easily accessible HTML documents. The implementation is also eased by using D3.js, which is a library for creating visualisations from data.

When the bar chart implementation was completed, the Data Visualisation solution was extended with an overlay visualisation, where the HTML elements of the webplayer are styled with animations that show the difference in activity between the two user groups.

As described above, there were eventually three implemented solutions. Figure 5.1 gives an overview of the work leading to these.

5.4 Development process

During the planning phase it was decided to take an agile approach to the thesis, ranging from the pre-study phase to developing the solutions and finally to writing this report. The work was conducted in two week sprints that were planned on Mondays. Each sprint had one or several large tasks, each with an explicit definition of done. This helped to concentrate the focus on the big picture for the sprint. Additional sub tasks were created to fulfill the larger ones. In order to keep track of the work, a web application called Trello was used.

Each morning a stand up was held before the work began. During these 10 minutes, the work of the previous day and the main focus of the current day was discussed. Additionally, every week was concluded with a 30 minute feedback session where certain key areas such as results, highlights, cooperation, lessons learned and what could be improved upon until next week were discussed.
Chapter 6

Results

This chapter describes the resulting implementations of this thesis. First, the functionality is outlined for the MapReduce jobs that extract the data. Subsequently it is described how this data is processed and used for each one of the different implemented solutions.

6.1 Data collection

The foundation of this thesis is the large amount of data on user behaviour in the client. In order to extract the required data from HDFS, MapReduce is used. The MapReduce jobs are written in Python to be executed with the Luigi framework. As the implementation of the solutions requires different data, two separate MapReduce jobs are developed. Figure 6.1 shows how these jobs consist of two chained subtasks each and how these are mapped to the implementations. Although the MapReduce jobs have different outputs, their input is the same. This input data consists of the log message from the web player client that is mentioned in section 3.2.1. The log message contains information about time, platform, source and some additional fields to describe the event that triggered the log message.

![Figure 6.1: Overview of the MapReduce jobs and their applications.](image)
6.1.1 Web player events

The WebPlayerEvents MapReduce job extracts the events related to the web player from the input source described in the section above over a given period of time. Each event belonging to the web player is compared to a blacklist of events that are not of interest, and in case of a match is discarded. Attributes of the event message that are not of interest are dropped as well. Output of the job is a table where each row is an event performed by a user in the web player. The table is first sorted on username and then on date. An example of the truncated event message and the output is presented in table 6.1. The Model Generation solution in section 6.4 uses this data as input as well as the WebPlayerDistinctCount as shown in figure 6.1.

WebPlayerDistinctCount calculates the number of times each event has occurred. The event count is incremented for a specific event if the context and event name matches. An event name is calculated by concatenating the Context and the Event attributes. The output is a list of each distinct event name together with the number of occurrences.

<table>
<thead>
<tr>
<th>Username</th>
<th>DateTime</th>
<th>Source</th>
<th>Context</th>
<th>EventAction</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Date1</td>
<td>Spotify:app:artist</td>
<td>artist</td>
<td>unfollow</td>
<td>hit</td>
<td>JSON</td>
</tr>
<tr>
<td>Bob</td>
<td>Date2</td>
<td>Spotify:app:context-action</td>
<td>context-menu</td>
<td>star</td>
<td>hit</td>
<td>JSON</td>
</tr>
<tr>
<td>Alice</td>
<td>Date3</td>
<td>Spotify:app:artist</td>
<td>artist</td>
<td>follow</td>
<td>hit</td>
<td>JSON</td>
</tr>
</tbody>
</table>

Table 6.1: An example of the output from the WebPlayerEvents MapReduce job.

6.1.2 Test user statistics

The main objective of the test user statistic job is to count event log messages that are triggered by test users and regular users respectively in order to be able to compare these two counts. This is a way to measure the test coverage of events in the Spotify client and this metric is referred to as event coverage from now. The first task of this job is ClientEventsPerUser. As its name suggests it sorts out events in the client and groups them by user. This process starts with mapping the username of each log message to the name of the event. The output records of ClientEventsPerUser then consist of a username and a list of the names and sources of the events that the user has caused. These records then constitute the input for the TestUserComparison task. Here, the unique test users and regular users respectively are first counted. The mapper then extracts the events from the lists in the input records and maps tuples of event sources and event names to a boolean parameter that depicts whether the user is a test user or a regular user. This parameter is determined by comparing the username to a list of usernames belonging to known test users. The reducer of TestUserComparison then receives a tuple representing a unique type of event with a corresponding list of boolean values. This way the occurrences of a specific event for the two user groups can be counted easily. To be able to compare these event occurrences it is appropriate to normalise each of the sums. This is done by dividing them by the total number of unique users of the group that have been active during the period measured on. The final output of the job is then the source and name of the event, together with the normalised counts of occurrences within the two user groups.
6.2 Data visualisation

The data collected with the MapReduce jobs is presented in its raw format as tabular data. In order to make it more accessible and simpler to draw conclusions from the data, two solutions are developed for visualising the data. The data that is visualised is the test user statistics presented in section 6.1.2.

6.2.1 Plotting

By utilising the Javascript library D3.js, bar charts from the data are constructed, as can be seen in figure 6.2. The plotting solution is built on Javascript and HTML and is accessible by any modern browser. It can either be hosted on a web server or run locally. The data is read from a tabular text file which is generated by the TestUserComparison MapReduce job described in section 6.1.2. A bar chart is then created and displayed for each app in the data. Each pair of bars represent the normalised frequency of the event label occurrence among regular users and test users. Figure 6.2 demonstrates that there are several events which are not triggered by the test users within the Artist app.

![Figure 6.2: Bar chart representing the differences between regular users and test users.](image)

6.2.2 Overlay

The overlay visualisation styles the HTML elements of the webplayer with blinking border animations depending on the difference of usage between regular users and test users. A rectangular border is animated around the element with variable size, color and intensity depending on the difference mentioned above. An example can be seen in figure 6.5. The animations are made by utilising the CSS transition support in modern browsers.

Spotify’s web player is modularised and consists of several apps served through iFrames on different domains. Due to the security restrictions imposed on iFrames during runtime the code have to be injected into the apps when they are first requested by the user.

A high level architectural view of the solution can be viewed in figure 6.3. After the data is extracted from the Hadoop storage using the MapReduce jobs in section 6.1, its values
Figure 6.3: Architectural overview of the overlay visualisation.

Figure 6.4: How output of the MapReduce job TestUserStatistics is converted into CSS properties.

are converted to CSS selectors and properties by a python parser as seen in figure 6.4. These values are then stored as JSON in a file awaiting to be injected. Each event name gets translated into a CSS selector for the specific UI element in the web player. In figure 6.4 click-album-related is the event name which gets translated into the matching CSS selector for the element which generated this event in the webplayer. Assignment of the color, border and secs values for each element is calculated by the difference between the numerical values in the event data, which are further explained in 6.1.2. The idea is that larger differences should generate faster animations, larger borders and contrasting colors.

The injector works as a middle layer between the apps and the web player. When a page in the web player is requested, the injector reads the JSON file and injects the css rules as a stylesheet into each app. When the page renders the modified stylesheet it changes the styling of the elements to reflect the test coverage according to the description above.

Since this solution is primarily meant for use by testers within the organisation the solution is deployed in a testing environment and not in production. This also results in that the minor increase in load times due to the injector is acceptable.
6.3 Test Runner

The test runner solution takes the most common user events generated from the webplayer as input and test their functionality. This enables the organisation to easily prioritise which parts of the webplayer should be tested. An overview of the solution is shown in figure 6.7 and explained in detail below.

The data made available by the MapReduce jobs is essentially a list of unique events and their frequency, for further explanation see section 6.1. The data is then parsed and converted to objects which are understood by the ClientEventTesting class. The ClientEventTesting class is responsible for starting the web player instance and executing tests against it. This is achieved by utilising the Spotify testing API which starts a browser locally on the computer running the tests and handles execution of the tests.

![Figure 6.5: A screenshot from the Spotify web player with color animated elements which highlights weak test coverage.](image)

```java
@AppTest(appName = "spotify:app:artist")
public class ArtistFrameTesting {
    @ClientEventTest(eventName = "artist/click-related-artist")
    public static void t_clickRelatedArtist() {
        <implementation>
    }
}
```

![Figure 6.6: How an event is converted to a test method.](image)
As previously described in section 6.2.2, the web player is divided into apps. In order to test an app, a class for the app containing the test methods has to be created, e.g. ArtistFrameTesting in figure 6.7 which encapsulate the test methods related to the Artist app. Each test method calls methods in the Spotify testing API which interact with the browser instance. The test methods also verify the return values from the API and assign a pass or failed value to each test method.

The translation between the events in the data to which test method to invoke is done via annotations. Each test method is annotated with the corresponding event name in the data. In the same manner each app class is annotated with the corresponding event source. Figure 6.6 gives an example of this.

![Diagram](image)

**Figure 6.7: Architectural overview of the Test Runner.**

6.4 Model generation

The objective of the Model Generation solution is to generate a GraphML file describing the most common event sequence among a cluster of users. This is achieved by clustering the user sessions based on similarity and from each cluster calculate a centroid which is finally converted to a GraphML file. The input data is explained in section 6.1 and an overview of the solution is shown in figure 6.9.

A session is defined as events generated by the same user and where the duration between
any two consequent events is no more than a predefined constant, in this case 300 seconds. In order to split the input data into user sessions, a script iterates through the data and creates sessions of events based on the session definition above. Sessions that contain less than five events are discarded. The sessions are represented as lists containing each event in the session.

To be able to organise the sessions into clusters, using the k-means method, they need to be represented as vectors. The values of these vectors should reflect the characteristics of the sessions that are desired to categorise with regard to. For this thesis, the interesting properties of the sessions are the most frequent events and the order in which they occur. Therefore, the Markov chain is chosen to model the sessions. One chain for each session is created and represented as a $n \times n$ transition matrix, where $n$ is the number of unique events in the data set. The structure of the matrix is the same as in 2.1. The values in the matrix are the transition probabilities between two events. To obtain a vector from the transition matrix, it is simply compressed so that each row is succeeding the previous. The resulting vector thus has the length $n \times n$.

With the vectors available, the sessions can now be clustered with regard to how close to each other in the $n \times n$-dimensional space they are. However, as mentioned in section 2.3.2, the k-parameter has to be specified in advance, in order to use the k-means method. As it is not known in prior what value $k$ should have for the algorithm to perform well, it is tentatively executed a number of times with different $k$ values. The result of this execution, containing values for $k$ in the range $[2, 25]$, is shown as a graph in figure 6.8. To maximise the tightness of the clusters, it is desirable to have an SSE value as small as possible. As seen in the graph, the value of the SSE decreases as $k$ increases. For large values of $k$, however, the SSE drop-off is diminished and the algorithm gets more computationally expensive. Therefore, the number of clusters need to be somewhat limited. At the same time, the silhouette coefficient, that is supposed to be as close to 1 as possible, helps to choose an appropriate $k$. From the measures in figure 6.8, $k = 10$ seems to be suitable for this data set, as the SSE drop from the previous $k$ value is comparatively large and there is an increase in the silhouette coefficient.

![Figure 6.8: The SSE and silhouette values for different k values.](image-url)
As the number of clusters is chosen, the result of this execution can be used to analyse the patterns identified by the algorithm in the data set of user sessions. From the resulting clusters, models can be generated to represent them. The centroid of each cluster is then extracted from each cluster by computing the element-wise mean of all the vectors that are members of the cluster. This centroid vector can easily be converted to a transition matrix. Then, this matrix is parsed by an algorithm that creates a session, i.e. a sequence of events, from it. This is done by first identifying the event with the highest probability of initiating a session in the concerned cluster. From there, the algorithm finds the event that is most likely to occur afterwards. This process is then repeated while each of the events until the probability of ending the session is the highest. The events that are selected constitutes the centroid session. The sessions generated are finally written to GraphML files. This is done by using data structures and output methods automatically generated from an XML Schema document that describes the desired structure of the XML-based GraphML file.

![Diagram](clusters.html) model.graphml

Create GraphML

Calculate centroid

Cluster sessions

Sessions to Markov chain

SessionSplit

Data

MapReduce

HDFS

*Figure 6.9: An overview of the Model Generation solution.*
Chapter 7

Discussion

This section presents a discussion of the results of the thesis. This discussion starts from
the three research questions in section 1.4.1, here referred to as RQ1, RQ2 and RQ3, and
then links these to the implemented solutions. Some advantages and disadvantages of the
solutions are also described, as well as their possible applications. Furthermore, the data
used for the analysis is discussed.

In RQ1, it is asked whether the testing at Spotify is relevant with regard to the behaviour
of real users in the system and if this is not the case, how the testing process might be
improved in this sense. To answer these questions, the Data Visualisation solution is
implemented. This solution consists of two separate implementations for visualising how
the SUT usage differ between test users and regular users. First, there is the bar chart
visualisation. As an example of this, figure 6.2 shows the usage of the events in the Spotify
Artist app. From these charts it is particularly easy to see which events that are not
triggered at all by test users although the regular users obviously use them. It is then
possible to draw the conclusion from this that no functionality is implemented in the test
automation framework of Spotify, that makes it possible to test these features. This might
depend on that the feature is newly implemented and therefore the test infrastructure
has not yet been implemented. A possible application for these bar charts is therefore to
identify new functionality that needs to be implemented in order to keep the test framework
updated and eventually maintain the event coverage.

The bar charts can also provide a useful overview of how the event coverage differ between
different apps. In some cases it is possible to see that this coverage is dependant on the
presence of test automation engineers in the teams that own the apps. Therefore the charts
might provide a basis for discussion about how resources need to be allocated.

The second implementation of the Data Visualisation solution is the CSS overlay which
visualises the same data as the solution presented above but in another manner. An
advantage of this approach is the mapping of event names to the actual components in the
web player that generated them. Understanding which component generated an event by
the event name alone requires specific domain knowledge of the logging framework. While
this information can be obtained by the viewer, it is subjected to the viewers interpretation
and might cause confusion due to misinterpretation. Presenting the data in this format also
enables the viewer to explore it in a familiar environment. The drawback compared to the
bar chart approach is the inability to quickly lookup events that need specific preconditions
fulfilled in the client, e.g. performing actions to generate the event. The bar chart approach
also enables the viewer to quickly compare the values of events due to the presentations of the bars, something that the overlay lacks.

Both implementations offer a solution to the same problem, namely to visualise the difference in usage of the web player. Whichever is more appropriate depends on the context. Roughly speaking, it boils down to speed found in the bar chart solution versus intuitive browsing as in the overlay.

A reasonable question related to the Data Visualisation is whether the event coverage is a good metric to reflect the test coverage of the SUT. This question is largely based on how well the client event log messages used as input data throughout this thesis describes the user behaviour. Thus, all three implemented solutions are affected by this issue. The log messages are chosen as input since they seem best suited, of the data available at Spotify, for solving the problem of the thesis. However, there are some problems with this data. For example, this type of logging is not very uniform, especially if it is desirable to model the user behaviour in the same way on different platforms. Although work is being done to make improvements in this area, it is not a priority issue. Furthermore, not all actions performed by the user are logged. To be able to model the behaviour of the user as accurately as possible, the logging needs to be more extensive than it currently is.

RQ2 asks how it is possible to model user behaviour. To propose an answer to this, the Model Generation solution is developed. The approach of this implementation is to create user sessions from event log messages and group these into clusters. In order for this clustering to provide a useful result, the session representations need to reflect the characteristics assumed to be interesting. As mentioned earlier, these characteristics in the case of this thesis are the most frequent events and the order in which they occur. Thus, it is of interest to study uninterrupted sequences of user actions. This is achieved by cutting off a session when the user is inactive for more than 300 seconds.

As mentioned in chapter 4, one risk of using a vector-based clustering method like k-means is that information is lost when converting the objects to categorise into vectors. In this thesis, the main information lost is the length of the sessions that might vary widely. However, this information is not very relevant for the purpose of generating test sequences. Therefore, the simple and intuitive k-means method is adequate to use for the clustering, rather than one of the more advanced approaches introduced in chapter 4.

When the sessions have been clustered, each of the clusters need to be represented as models. This problem is solved by extracting the cluster centroids and from each of them produce the most probable sequence of events. The way this algorithm is currently implemented, it does not take the length of the cluster session into account. This is not a problem per se, but it sometimes results in centroid sessions being cut off too early and thus missing out important events. Therefore, this algorithm might need to be slightly modified.

It is also relevant to question the abstraction level of the input data. The logging messages that are extracted reflect functionality on a slightly lower level than the models used for the MBT testing at Spotify. Besides, it is questionable that only one data source is used. There might be a possibility to combine the chosen log messages with other data on different abstraction levels in order to obtain a more representative model of the user behaviour. However, this would also result in a more complex data processing and implementation.

The Test Runner implementation described in section 6.3 is developed in order to investigate the question posed in RQ3. Because of the data used as input to the Test Runner
consists of how GUI components are used, the testing in this scenario is limited to system testing of these components. However, the Test Runner is not tied to system testing specifically but can also be used for testing on lower or higher levels. In this case the Spotify testing API and the data used for input decided which level the testing should be performed at. If those two factors are changed, the Test Runner can perform testing on another level.

To utilise the Test Runner to its full extent, it is meant to run automated tests in a regression context. The value lies in the prioritisation of test cases done by the underlying data analysis depending on the amount of usage. This is important in environments where test execution is expensive due to limited resources or time consuming tests for example. However, in the case of Spotify this is not the case since their test execution is cheap.

In order to successfully implement a test automation system, it needs to be maintainable and this is the reason annotations are used for mapping events to test methods. Annotating the test methods as described in figure 6.6 means that there is only one location where the test developer needs to alter the code when implementing new tests or modifying older ones.

The above discussion describes the implemented solutions, their applications and benefits versus drawbacks separately. However, it is also interesting to explore the possibility of using combinations of them. This could be applied for example by first using the Model Generation to create models of the most common user behaviour. These models can then form input for the Test Runner if it is extended to be able to execute client event tests not only separately but also in a series. This way, the most common sequences of user actions can be tested. Additionally, the Data Visualisation can be used to get quick feedback on where more testing needs to be implemented in order to maintain the test automation framework.

7.1 Future work

The intersecting area between software testing and data-driven analysis is an interesting and emerging topic. While this thesis discusses some important issues surrounding the topic there are still issues that would be worth investigating further.

One such issue is the type of testing performed with data augmentation. Perhaps other types of testing such as performance testing or acceptance testing are suitable for augmentation by data. A question related to this is whether the data used in this thesis for functional testing would be suitable for this purpose or if other data sources are required.

Another issue worth investigating further is the use of other data mining algorithms. The algorithms presented in this thesis are simple and intuitive, meanwhile, the current research in user behaviour modelling presents more advanced solutions that can capture more information and make more accurate models. Investigating other use cases for the developed solutions is also interesting. One such use case could be to measure the energy consumption by an app such as Spotify on mobile phones. By generating test sequences of the most common behaviour and executing them while measuring the energy consumption, behaviour which drain the battery could be quickly identified and corrected.

Focusing on the presented solutions, the largest benefit to be made is by automating the entire toolchain. This would mean that the MapReduce jobs are run periodically against
the data and imported into the solutions which also are run continuously. Overall, this will likely increase the usability and therefore usage of the solution by minimizing the amount of manual work required. Additional improvements could be made to the model generation as well. As of now the models generated consists of straight sequences but could be extended to support recursive edges or diverging paths by allowing more than one edge to originate from an vertex. This would enable the models to describe the SUT in more detail.
Chapter 8

Conclusions

In this section of the report, the research questions are once again brought up to be answered.

After implementing the Data Visualisation solution and viewing its results, it is possible to give the answer to RQ1 that there are some differences between how the Spotify web client is used and how it is tested. The Data Visualisation solution is an attempt to provide feedback on where additional testing needs to be implemented. As discussed earlier, other possible improvements of the testing process are also proposed in the Model Generation and Test Runner solutions.

Regarding RQ2, the Model Generation is one attempt to solve the problem of modeling common patterns in the behaviour of the SUT users. However, there are alternative ways to do this, some of which are discussed in chapter 4.

RQ3 asks how user behaviour data can form input for test automation. The Test Runner solution suggests one way of doing this, by parsing event logging messages and mapping them to test implementations. One possibility is also to take the sessions created by the Model Generation solution as input, in order to execute tests of the most common sequences of user actions.

8.1 Implementation status

The current status of the implemented solutions is that they are prototypes that demonstrate different ways to use data for augmenting test automation. In the case of the Data Visualisation, the bar charts are ready for deployment, while the overlay prototype needs some more mappings between client events and CSS selectors to be fully functioning, as only a few are implemented currently. Besides, threshold values for which events to animate need to be set. The Model Generation solution is working but it can be modified with additional data mining algorithms, as mentioned in section 7.1. For the Test Runner, more tests need to be implemented, but otherwise it is fulfilling its purpose to execute regression tests. All three prototypes have been handed over to Spotify together with the documentation needed to run them.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmännen medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmännens ideella rätt innefattar rätt att bli nämnda som upphovsmän i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmännens litterära eller konstnärliga anseenden eller egenarter.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owners. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the authors have the right to be mentioned when their work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Erik Kurin & Adam Melin