EXAMENSARBETE INOM DATATEKNIK,
GRUNDNIVÅ, 15 HP

*STOCKHOLM, SVERIGE 2016*

# Evaluation of tools for automated acceptance testing of web applications
# Utvärdering av verktyg för automatiserad acceptanstestning av webbapplikationer

**BASHAR AL-QAYSI**

**SARA BJÖRK**

**KTH**
**SKOLAN FÖR TEKNIK OCH HÄLSA**

# Evaluation of tools for automated acceptance testing of web applications

# Utvärdering av verktyg för automatiserad acceptanstestning av webbapplikationer

Bashar Al-Qaysi
Sara Björk

Examensarbete inom
Datateknik,
Grundnivå, 15 hp
Handledare på KTH: Anders Lindström
Examinator: Ibrahim Orhan
TRITA-STH 2016:49

KTH
Skolan för Teknik och Hälsa
136 40 Handen, Sverige

## Abstract

Auddly provides a music management tool that gathers all information about a musical piece in one place. The acceptance testing on their web application is done manually, which has become both time and money consuming. To solve this problem, an evaluation on automated acceptance testing was done to find a testing tool suitable for their web application. The evaluation was performed by finding the current existing testing strategies to later compare the tools implementing these strategies.

When analyzing the results it was found that two testing strategies were best suited for automated acceptance testing. The Visual Recognition strategy that identifies components using screenshots and the Record and Replay strategy that identifies them by their underlying ID. The choice between them depends on which of these properties are modified more often.
It was also found that automating acceptance testing is best applied for regression testing, otherwise it should be performed with a manual approach.

It was made clear that the Selenium tool, which uses the Record and Replay strategy, was best suited for Auddly's acceptance testing. Selenium is able to test AJAX-calls with a manual modification and is a free and open source tool with a large community.

**Keywords**
Automation, acceptance testing, regression testing, single-page application

## Sammanfattning

Auddly tillhandahåller ett musikverktyg som samlar all information om ett musikstycke på ett enda ställe. Acceptanstestningen på deras webbapplikation sker manuellt, som både blir tidskrävande och dyrt. För att lösa detta problem har en utvärdering av automatiserade acceptanstestverktyg genomförts för att hitta det verktyg som passar deras webbapplikation bäst. Utvärderingen utfördes genom att hitta existerande teststrategier för att sedan jämföra de verktyg som implementerar dessa strategier.

I analysen av resultatet framkom det att två av strategierna var mest passande för automatiserade acceptanstester. Strategin Visual Recognition som identifierar komponenter genom skärmdumpar och strategin Record and Replay som identifierar de via deras underliggande ID. Valet mellan dem beror på vilka av dessa egenskaper som ändras oftare.
Det framkom även att automatisering av acceptanstester är mest lämpligt i regressionstestning, i andra typer av testning bör det ske manuellt.

Det klargjordes att verktyget Selenium, som använder strategin Record and Replay, var det bäst passande för Auddly's acceptanstestning. Selenium kan testa AJAX-anrop med en manuell modifiering och är ett gratis verktyg med öppen källkod samt ett stort forum.

**Nyckelord**
Automatisering, acceptanstestning, regressionstestning, single-page applikation

## Preface

This report is the result of a thesis work in computer science at KTH Royal Institute of Technology, on behalf of the company Auddly.

The authors of this report want to thank all of the helpful staff at Auddly for your support, especially Emil Wallinder and Johan Greus. Also a special thanks goes to our supervisor at KTH, Anders Lindström.

## Glossary

| | | |
|---|---|---|
| **AT** | - | **A**cceptance **T**esting is testing if the software system meets the requirement specifications. |
| **AUT** | - | After an application is designed and coded, it goes for testing and is then stated as **A**pplication **U**nder **T**est. |
| **GUI** | - | The **G**raphical **U**ser **I**nterface allows the user to interact with the underlying system by the use of graphical elements. |
| **SPA** | - | **S**ingle-**P**age **A**pplication is a web application that is contained on a single web page in order to create a fluid and responsive browsing experience for the user. |
| **SUT** | - | When a system is being validated through testing it is referred to as **S**ystem **U**nder **T**est. |

# Table of contents

# 1 Introduction

## 1.1 Problem statement

The testing of software is becoming more laborious and time-consuming as the software systems are getting more advanced. Any nontrivial software has at least one bug, and testing the system once is insufficient to ensure an error free system. Acceptance testing (AT) aims to establish if the system has met the functional requirements.

Auddly develops a music management tool that gathers all information about a musical piece in one place, as an independent bridge between the music industry's copyright owners and master owners. The AT on Auddly's system is done manually. Manual testing can become both time and money consuming, because it requires a person to interact with the graphical user interface (GUI) to find bugs.
To be able to work more efficiently, Auddly sees a great benefit in automating their AT. By automating the testing of a system, a repetition of an automated test becomes much more effortless after a modification of the source code e.g. when a bug is found and fixed.

There is a lack of knowledge in automated AT and since AT is such an important part of system developing, automation is usually dropped.
For this thesis different strategies for automatic AT of web-based applications was evaluated to find a tool that can replace the current manual testing. A comparison of automatic and manual AT is also presented.

## 1.2 Research goals

Auddly's system contains a web application where the GUI often changes. Thus an automatic AT tool that makes test maintenance easier was demanded. It was also essential that the tool is capable of testing not only traditional web application but also single-page applications. With the AT tool it should be easy to write test cases i.e. by providing a good documentation and examples for beginners. Other important criteria can be found in chapter 3.

The final goal of this thesis is to propose a suitable automated AT tool, meeting Auddly's requirements. To accomplish this, a research on the current existing testing strategies was made in order to find the applicable ones for the system. These strategies were evaluated to find the most suitable one for Auddly.
To achieve the final goal, the task was divided into smaller objectives.

- There are multiple implementations, tools, for each testing strategy. These tools will be compared with the ones implementing the same strategy to find the one that meets the most criteria for the strategy. The comparison will be accomplished by a practical execution and testing of the AT tools.

- A comparison of the testing strategies through the AT tools chosen will be performed to later find the best applied test strategy for Auddly's system. This comparison will also be done by practically testing the AT tools and comparing them according to the criteria mentioned in section 3.2.

- An analysis of when it is most effective to use automated AT in comparison to manual AT will be presented as a compilation of the evaluation.

## 1.3 Delimitations

The following areas will not be included in the thesis:

- Only web-based software will be tested with automatic AT.
- No analysis on cost of the different testing strategies will be performed.
- Auddly's entire system will not be tested.
- Only strategies with existing tools will be presented and compared. For every strategy, one existing tool will be chosen according to the criteria explained in section 3.1.

# 2 Theory and background

This chapter gives a technical background on automated acceptance testing that is necessary for understanding the aim of this thesis. This chapter also describes the theory of automated testing and currently existing testing strategies.

## 2.1 Acceptance testing

Despite how experienced the developer is, when software is built it is more likely to contain bugs. Hence software testing is one of the essential parts in the development process. Software testing is also the most expensive part of a system's life cycle. According to Memon [1] testing is often accounting 50-60% of total development costs. In 2003 the United States alone lost $59.5 billions [2, p. 2] because of bugs that testing tools won't recognize each year. Improved testing infrastructure should try to reduce the current software bugs by one-third thus saving $22.2 billions according to Li, Wu [2, p. 2].

Acceptance testing (AT) is a testing approach that is build upon the customer-developer handshake. Thus it has a high level of integration, usually between the user interface and the business logic [3][4]. AT is implemented to ensure the functionality of the overall application under test (AUT).

Most modern software applications are developed as web-based applications that can be accessed through a web browser. Therefore most of the present applications have a graphical user interface (GUI) that enables the user to interact with the underlying system. Haugset, Hanssen [4] believes that a rational place to generate AT would be at the GUI level.

AT can be implemented in two ways, manual or automatic. Manual testing can become time consuming because it requires a lot of effort from the tester while providing low coverage criteria. It can become difficult to determine if the GUI is even correctly tested because the decision will be based on the experience and sensibility of the person who is testing it. Pimenta [5] concludes there is a lack of qualified test specialists. Automating GUI tests is more efficient and time saving than the manual test if it is done correctly. An automatic test can run more often and in more ways, by using different sequences that would not be possible by manual testing. It is simpler to rerun a test automatically when a bug is found and fixed than manually because the tester doesn't have to remember the sequence of events that revealed the bug.

AT is a repeatable task usually done once or multiple times a day to detect new bugs in the application. The execution of AT is an interactive process between the user and the system under test (SUT). For a test to be automatic, the testing tool must be capable of simulating the inputs from the user on the AUT to a great extent [1]. This task can become very difficult but is solvable by using one of the existing testing strategies, presented in 2.3.3.

The aim of this thesis is to examine which one of these strategies is the best-implemented one for GUI testing web applications. This task will be achieved by analyzing a number of available testing tools.

## 2.2 Web applications

A presentation of the existing web application infrastructures, Auddly's web application and their current testing processes is done in the following subsections.

### 2.2.1 Traditional and single-page web applications

Traditional web applications usually consist of a user interface that contains multiple pages. Every page can be seen as a representation of a state [6]. Interacting with a traditional page cause a request to the server. The server then locates and returns a new HTML page for the client (browser) to display. This triggers a page refresh in the browser [7], as seen in figure 2.1a.

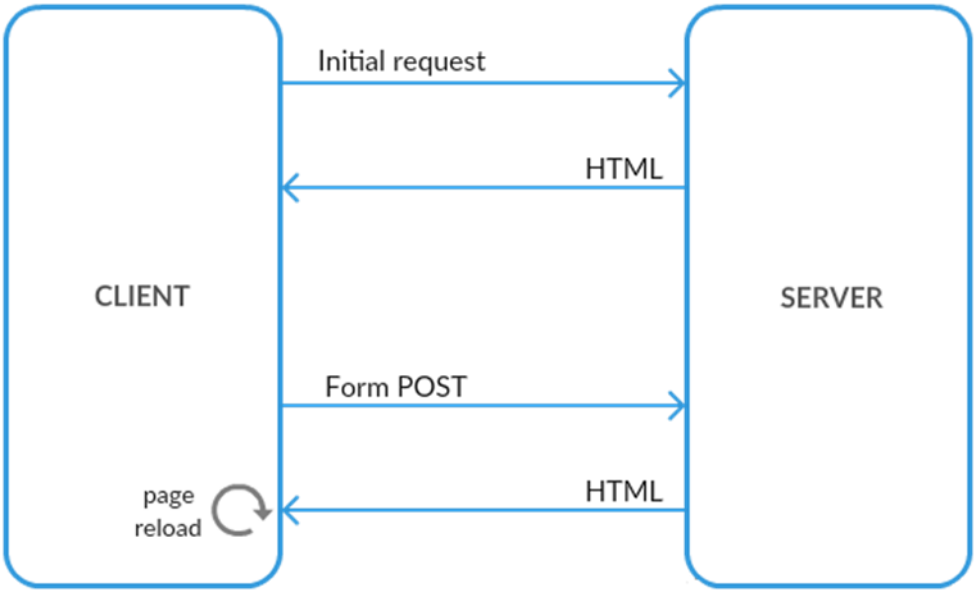


Figure 2.1a: How traditionl web pages work.

Single-page application (SPA) is a new web development approach that is constructed to build responsive web apps that consist of a single HTML page. SPA uses a technology named Asynchronous JavaScript and XML (AJAX) to build apps that are dynamically updateable by the user interactions [8]. The app contains different components that can be updated independently, hence no constant page refreshing occur. SPA uses JSON or XML data to update the page's components, see figure 2.1b. A number of SPA frameworks has been developed over the past years e.g. AngularJS by Google, which Auddly is currently using.

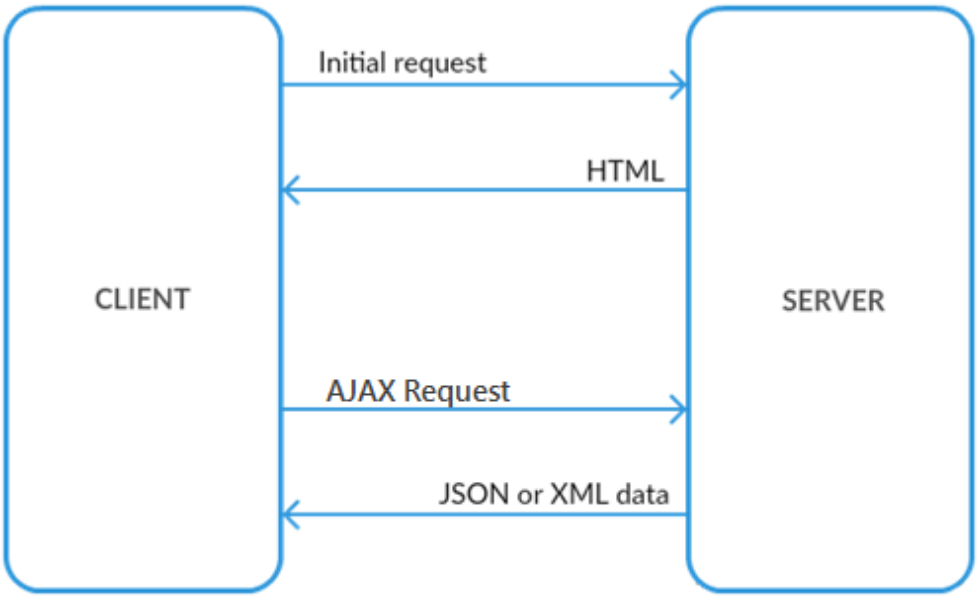


Figure 2.1b: How SPA works.

### 2.2.2 Auddly's system

Auddly maintains a single-page web application built with AngularJS, which is an open source JavaScript framework.
The acceptance testing of the application is currently done manually by a tester. The test cases are determined and documented in advance, and usually consists of registration, logging on to the system, etc.
When new functionality is added to the application, the tester validates the new scenarios, but the main purpose is to make sure the new functions didn't break any of the old ones. These are the functionalities that mainly require automation.
Auddly uses continuous integration for their unit testing. By automating their acceptance testing they could speed up the development life cycle even further by adding these tests to their workflow.

## 2.3 Automated testing

As mentioned earlier, testing is one of the most important parts in the software development process. By automating the testing process correctly, efficiency and time saving is guaranteed. This section describes different concepts related to software testing.

### 2.3.1 Test cases, test oracles and test suites

To test an application, at least one test case must be provided to execute. A test case is a number of related operations that run successively in order. In AT testing, test cases usually assemble user actions on the user interface, often a GUI (see Figure 2.2), to later check the result of the actions.

The analysis of the result obtained by a test case is usually done by a mechanism called a test oracle. The test oracle's task is to check if the current output of each test case is the expected one, as displayed in figure 2.2. A test oracle contains the expected output from a certain input to compare it with the actual output of the SUT, if the parameters are equal: the test is successful, otherwise the test failed. [9][10]

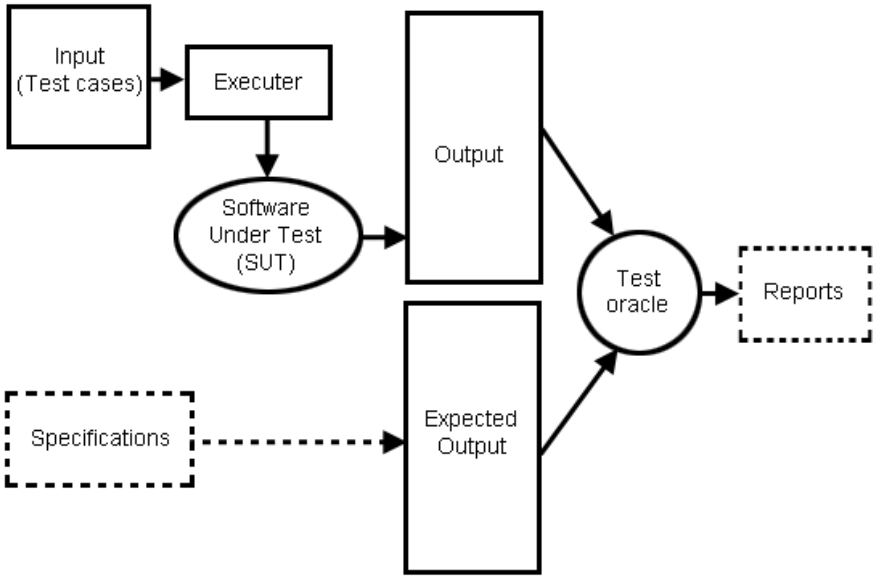


Figure 2.2: The process of test case execution.

A *test suite* is a combination of a small subset of the total number of available test cases. These test cases are selected and ordered according to a certain criterion of the SUT. The meaning of a test suite is for testing one scenario of the AUT.

### 2.3.2 Regression testing

Regression testing is one type of maintenance testing [12], which involves retesting a modified program. The new update of the code is tested with the old test cases. This is done to verify that the old functionality is still working. An update might cause errors in other functionalities of the application that wasn't intentionally changed.

When the AUT is updated in order to fix a bug or implement more functionality, a regression test can determine if the application contains new bugs. Regression testing also ensures that the AUT doesn't decrease in quality by the modified parts [11].

### 2.3.3 Current strategies

In this section, a representation of the different AT strategies that are concluded from the pre-study is displayed.

#### Record and replay

According to Li, Wu [2, p. 20] the most popular automated testing tools implement the Record and Replay (R&R) strategy.

The tester interacts with the UI, for instance pressing keys and mouse clicking, the tool records the mouse coordinates and stores them in a test script. The test is later performed against the SUT by re-executing (replaying) the test script.

The problem that can occur with mouse coordinates is that the test script may break at minor changes of the GUI. In modern R&R test tools this has been fixed by identifying the components as

objects by storing their properties instead of the mouse coordinates [2, p. 20]. Replaying the test will execute the events on these objects.
Kent [13] claims that this strategy isn't very automatic since the tester needs to manually create the test cases. It involves a continuous interruption of inserting verification points and the recording can become intensive. Even though, this technique is available in most modern test tools.

**Model-based testing**

Model-Based testing (MBT) is an automatic testing strategy that is dependent on using a software model for the creation of test cases. The model used in the testing is created based on the requirement of the SUT. It is normally written manually by the tester from the SUT specifications using a regular programming language like C# or using a modeling language like Unified Modeling Language (UML) [14].
Mäkinen [15] means that the creation of the model requires a lot of, most, effort from the tester because it demands a big knowledge in the SUT and the programming language used by the tool for creating the model. After the model is written, the MBT tool can then automatically create test suites from the model and run them to test the overall SUT. Another problem with this testing approach is that the written model must be available and updated in the system development process, not only in the testing process.

**Random testing - Test monkeys**

Test monkeys acts randomly without the tester's participation. A test monkey interacts with the application randomly by key pressing and mouse clicking. The monkey has no knowledge in how a human uses the application [2, p.22], hence it can find bugs differently than a manual test.
Test monkeys differs in intelligence, there are dumb and smart monkeys [5]. Dumb monkeys set up test cases randomly. They can't detect software errors, which is the reason why they aren't effective when searching for incorrect behavior. But they are more effective for finding sequences that crashes the SUT, which is their goal. They don't have any knowledge about the states of the application, but smart monkeys do. Smart monkeys also know about the legal steps for each state, unlike dumb monkeys. Smart monkeys are although more difficult to build than dumb monkeys since they require a state table.
Microsoft announced in 2008 that 10 - 20% [16] of their bugs were detected with monkey testing.

**Visual recognition**

Visual GUI testing (VGT) is a strategy that is much alike R&R but instead of coordinates it uses image recognition to interact with the GUI elements [17]. The bitmap interaction grants the VGT to imitate the user while treating the SUT as a black box. When connecting test scripts with image recognition, this strategy can be implemented on all platforms.
With the VGT tools the tester writes the test scripts manually and includes images of the GUI elements from the AUT. The images identify the inputs from the user and the output is then compared to the expected result. According to Börjesson, Feldt [18] the image recognition can cause sensitivity to changes of the GUI elements, such as change of color and position.

**Keyword-driven**

With the keyword driven strategy, the testing is done through step-by-step instructions. The instructions are performed with a table that documents the functionality of the SUT [19]. The strategy is for that reason also called table-driven.
The table is then mapped from the keywords to the actions that are executed in the test scenario. These test components can also be reused for other test scenarios.

Keyword driven tests can be performed in separate phases; a planning and an implementation phase. This is an advantage when persons with different competences do the testing. The planning phase is usually done by the person who knows about the functionality of the application but has no programming knowledge, and the implementation is then done by a developer.

# 3 Methodology

This chapter describes methods and evaluation parameters on the strategies to meet the goals defined in section 1.2.
The testing strategies and testing tools were found through a literature study and evaluated through manual testing. The evaluation of the different test strategies was done through choosing one testing tool from each strategy. The choice of tool, that implements a specific strategy, had to be based on the criteria in section 3.1. Auddly's testing web application was used for evaluating the tools. The strategy that fulfilled most of the parameters from section 3.2 was the best fit for automated testing of web applications.
The criteria presented in sections 3.1 and 3.2 were discussed with the developers working at Auddly. The discussion was made to further validate and uncover new criteria for the validation of the tools and strategies.

Alternative solution methods could have been to evaluate the testing strategies only based on the results from the literature study. A decision could be made on which testing tool would be the best fit by comparing the pros and cons from the study. The downside of this method would be the difficulties in drawing conclusions based on other's studies. The information would not be as reliable as creating custom made analysis and therefore had not been suited for this thesis.

## 3.1 Choice of testing tools

A theoretical research was done to find out how a tool accomplished the criteria in 3.1, by for example reading the tool's documentation. One of the criteria, single-page support, had to be practically tested on each tool in order to be evaluated.

### 3.1.1 The evaluation criteria

Some of the following criteria weigh in more heavily than others in the evaluation of testing tool. The criteria that aren't equally crucial contribute to the final decision if the tools are very similar. A tool that is active (support criterion) and capable of testing single-page applications are the two most important criteria.

**Single-page web application support**
In dynamic web applications it can be difficult for the testing tool to detect an update if no page refreshing occurs. For testing single-page application it is important that the testing tool supports testing AJAX calls in order to avoid timers so the test case knows when the result should be expected.
Does the tool provide a testing functionality for single-page apps or does the tester have to write own solution that is designed primary to the system under test (SUT). If a testing tool is capable of testing a single-page web application, it can also test a traditional web application.
This method is further described in 3.1.2.

**Documentation**
To replace the current manual testing with an automated testing tool it would be easier to get started with a well-documented tool.
How well the testing tool is documented, does the documentation contain examples for beginners. Open source system will be good enough since the source code is available to read, although it would take longer time to learn.

**Web browser compatibility**
Modern web application run differently on different web browsers. Thus the acceptance-testing (AT) tool should be able to test the SUT using different browsers.

**Environment**
In order to test the application in different environments it is important that the tool is compatible with different operating systems. It would also benefit the tester not having to learn a whole new programming language to be able to create a test suite.
Which operating systems the testing tool is compatible with or which programming languages it requires writing testing scripts.

**License costs**
How much does the tool license cost. The costs will be compared only for the benefits of Auddly. Free and open source tools are preferred.

**Support**
How old is the tool and if the tool is currently maintained and not discontinued.

### 3.1.2 The practical evaluation

The testing of single page support was made on Auddly's testing web application. Two different tests cases were made for each testing tool. The first test was made to check the tool's capability of testing a traditional website. The second test case was made to the tool's capability of testing single page applications.

The first test case consisted of multiple steps to login to the website (providing username and providing password and clicking the sign in button as seen in figure 3.1) then verify the outcome.

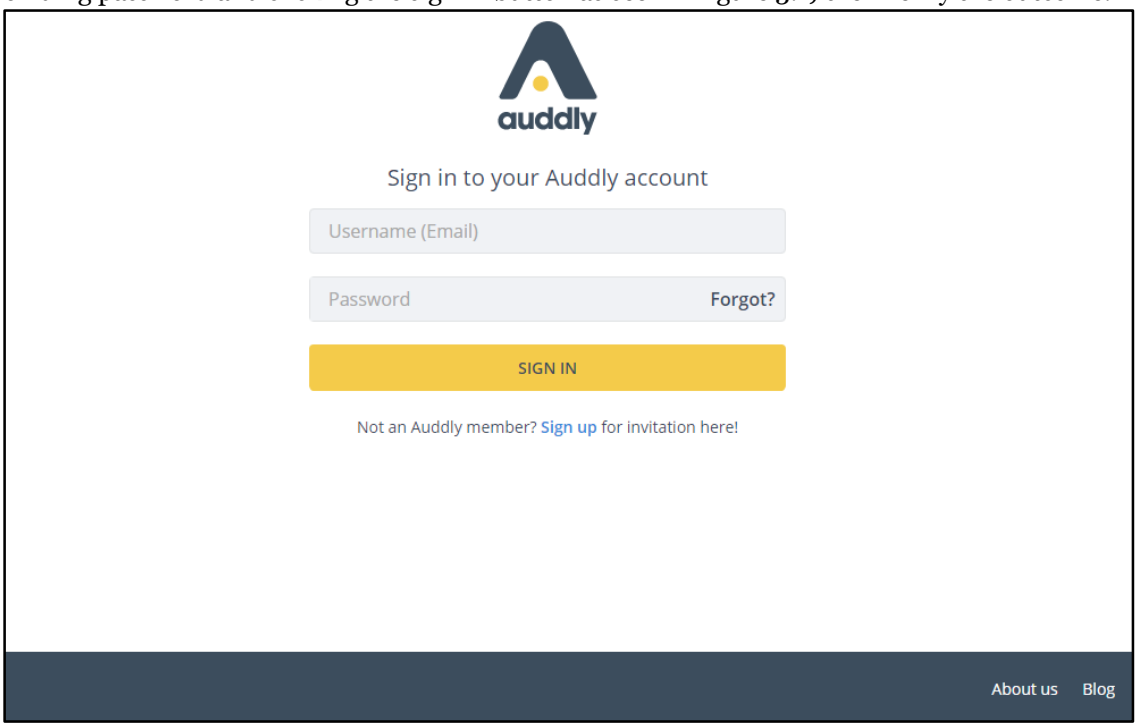


Figure 3.1: Auddly's testing web application, the login page.

When a user successfully logs in, he is transferred to the songs page, see figure 3.2. To verify the login test case, a test oracle was created to check if the profile name (DJ BASHAR) exists as highlighted in figure 3.2.

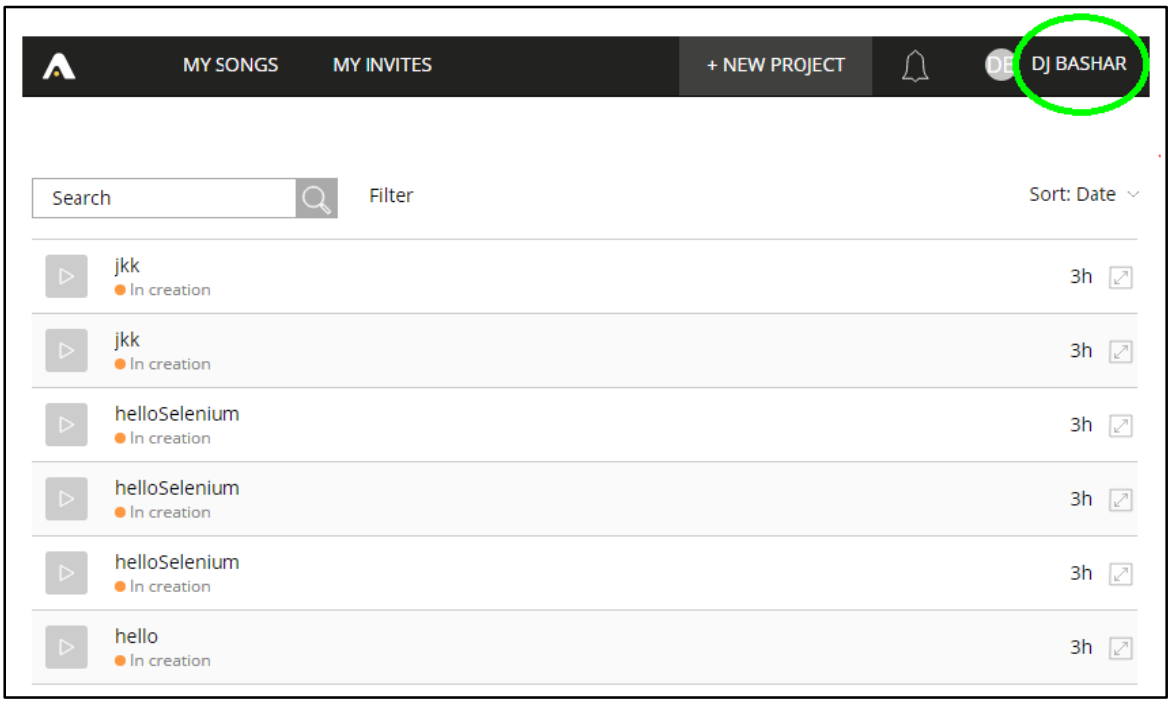


Figure 3.2: Auddly's song page.

The second test was made by creating a new project in Auddly's web application. The testing tool would click on the "+ New Project"-button to be redirected to the create page, see figure 3.3. In this page a name of the song project would be entered and both the checkboxes clicked.

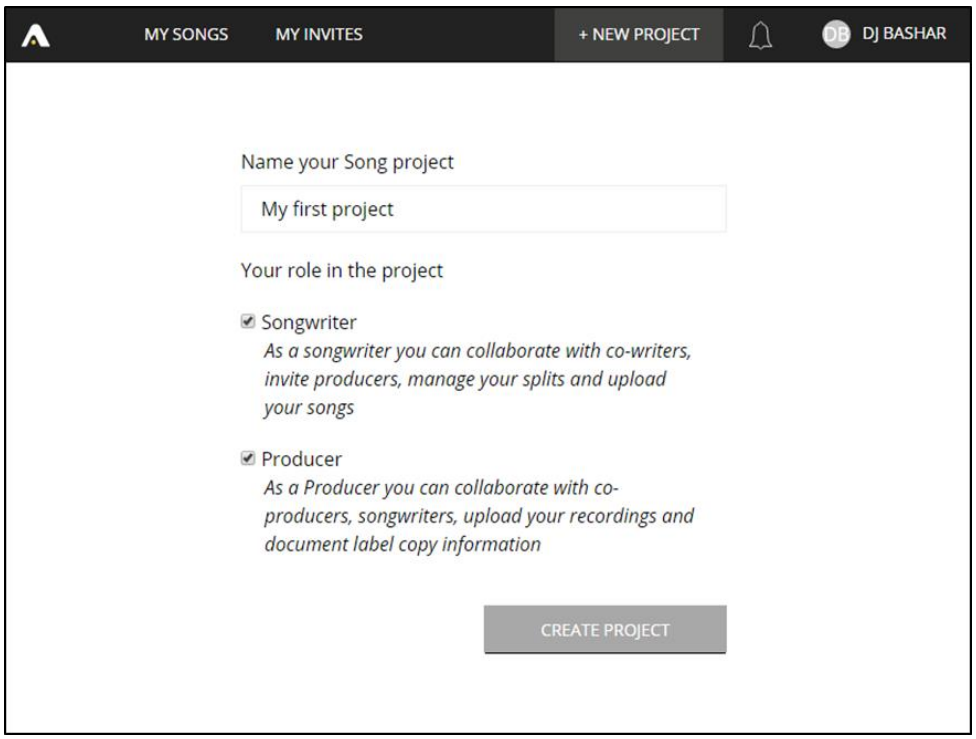


Figure 3.3: Auddly's create new project page.

After pressing the “Create Project”-button the tool would redirect to the project’s page. The test oracle checks if the name of the song project entered in figure 3.3 matches the name in the highlighted header in figure 3.4.

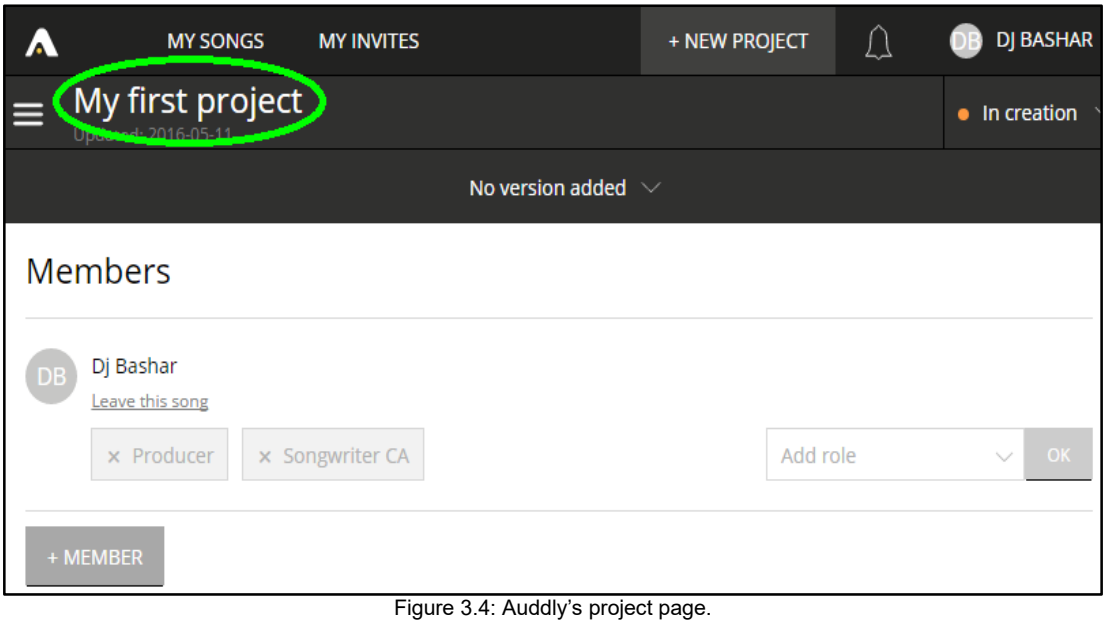


Figure 3.4: Auddly’s project page.

## 3.2 Strategy evaluation parameters

The following parameters were used to determine the best strategy for testing web applications. To evaluate if a strategy could check the outcome of a test case and how the maintenance of a tool, was done by a theoretical analysis. The evaluation of the first mentioned criterion (handling GUI) was done by manually testing the tools (that were the result from 3.1) before and after a change on the AUT with the same test cases. The tests were performed on the same web application as Auddly performs their current testing on, since it required modification of the code.
All of the following criteria will be taken in consider when analyzing which strategy is the best suited one for automating acceptance testing of web applications.

### Handling the graphical user interface (GUI)

Since AT is mainly about testing the functionality of the application under test (AUT), a testing tool that is not affected by GUI changes is required. For example, a test case shouldn’t fail if a button’s position has changed.
This criterion was tested by changing the position and the color of a component in the application, and observing if the test case failed.

### Maintenance

The maintenance of a testing tool can be a very difficult task due to the fact that maintaining a test is dependent on the capability of the tester to write effective test cases and at a right level of details. A test suite is usually easier to create than to maintain, that is because of the number of different ways a test case can fail.
This was evaluated through a theoretical analysis on how the strategies can maintain a test case.

**Verifying test results**

The chosen AT tool should be able to verify the outcome of a test case through one or more test oracles. This is for the tester to know if the test case failed or succeeded. This criterion was verified by a theoretical analysis of the strategy's support for test oracles and by further testing of the tool.

## 3.3 Evaluated strategies

A number of testing strategies were presented in section 2.1.4. The model-based strategy was not tested though. This decision was made due to the model's complexity and its low learning curve; it requires, as mentioned earlier, knowledge in a modeling language and a basic understanding of the SUT to write the testing model.
The other strategies mentioned was compared and analyzed as planned.

## 3.4 Comparison of manual and automatic AT

As an ending stage of this thesis, a comparison of manual and automatic AT will be made. This goal will be achieved by a theoretical analysis based on the outcome of this thesis. The result will be presented and further detailed in chapter 6.

# 4 Results

The result of this work is divided into smaller subsections to fulfill the goals mentioned in section 1.2.

## 4.1 Testing tools

Below follows a summary of the testing tools for acceptance testing (AT) that was tested. The full result of the study is found in the Appendix A.

### 4.1.1 Random testing

**gremlins.js**

A “dumb” monkey testing framework that is compatible with all existing browsers. Gremlins.js have support for single-page applications, but is however incapable of generating logs and reports.

Without any alternative choices the gremlins.js testing tool was chosen for the random testing strategy.

### 4.1.2 Record and replay

**Selenium**

Selenium automates testing of web applications by recording test cases in a FireFox web browser and replaying them in any other browser. Selenium WebDriver supports dynamic websites natively, meaning a small manual edit is required for testing single page applications.

**Sahi**

Sahi records and replays test cases in any web browser, on any platform. The tool supports single page applications and records Sahi Script that is based on JavaScript.

Selenium was chosen over Sahi as the testing tool implementing the record and replay strategy.

### 4.1.3 Visual recognition

**Sikuli**

Sikuli automates tests using image recognition to control graphical user interface (GUI) components. It works on all browsers and on any platform with support for Java.

**RoutineBot**

RoutineBot is a visual test automation tool that creates test scripts with image recognition. It requires Windows to be used but isn’t limited to any browser.

**Automa**

Automa is a GUI automation tool for Windows with image recognition functionality. Automa can be used with any IDE that supports the Python programming language.

Sikuli was chosen as the testing tool implementing the visual recognition strategy.

#### 4.1.4 Keyword-driven

**Ranorex**

Ranorex creates keyword driven tests with a record and replay functionality. The tests are recorded on a Windows platform but can be run on most common browsers, such as Firefox, Chrome and Safari.

**Silk Test**

Silk Test is a keyword driven tool that supports cross browser testing on Windows. The tool can test single page applications but the lack of documentation on their website decreases the usability.

**TestComplete**

TestComplete provides record and replay functionality for creating test cases. The tool builds keyword driven tests and automates testing of single page applications. The platform provides many extra functions, which is why it's pricey.

The tool chosen implementing the keyword-driven strategy was Ranorex.

## 4.2 Testing strategy

The testing strategies were evaluated with the parameters from 3.2. The results are presented below.

#### 4.2.1 Handling the GUI

**Random testing**

In random testing, only functionality is tested. The tests are randomly generated and don't compare any expected result with the current. Therefore the strategy isn't affected by GUI changes.

**Record and Replay**

With the record and replay strategy the elements can be located with different types of locators. The performance on this criterion depends on the type of locator. These can be modified in the test for desired behavior.
If the locator identifies the element by its' ID, the strategy is not affected by any changes whatsoever of the GUI, whether it is a new color or a new position of a certain element.
If the locator is by CSS, the strategy binds the style properties of the element. This can cause the test to break if the positioning of the element is changed.

**Visual Recognition**

The visual recognition strategy fails when a modification in appearance of a GUI element is done, e.g. change of the color. Even though this strategy is based on the graphical identification, it is not affected by the change of a GUI element position if the compared image is specifically referring to this element.

**Keyword-driven**

This strategy could successfully find and select the elements after a change of the GUI has occurred. It had a similar approach of identifying an element as the record and replay strategy.

### 4.2.2 Maintenance

**Random testing**

No maintenance is required with random testing. This strategy doesn't contain any test oracles, hence it doesn't depend on any previous test suites or on the application that is tested.

**Record and Replay**

There are two ways to correct a test case failure, e.g. due to a change in certain functionality. Either by a manual change of the test case by modifying the step where the test case failed, or recording the new functionality from where the test case failed.

**Visual Recognition**

In the visual recognition strategy, an element is recognized by a screenshot in the test case. If a modification in the GUI has been done, all the tester needs to do is to take a new screenshot of the new element. Neither the position of the element or its' functionality is specified in the test case.

**Keyword-driven**

For the keyword-driven strategy, problems arose with the Ranorex tool when testing Auddly's system. Ranorex was found to not work properly. The tool wasn't at all times able to find the elements that were present on the screen. Because of this, the tests couldn't be run as planned.

### 4.2.3 Verifying test results

With random testing there is no way for the tester to verify the results of a test, besides knowing that the application can crash but not in which sequences the crash occurs.
There is also no way to predict at what time it detects a bug, and is therefore difficult to establish at what time it is done testing the application.
With the remaining strategies it is possible to create test oracles to verify the output of the test case.

# 5 Analysis of the result

An in depth analysis of the result of this work is presented in the following subchapters.

## 5.1 Choice of tools

The tools evaluated for the record and replay were Selenium and Sahi.
When Sahi's IDE was tested, it wasn't as stable as expected. When replaying recorded scripts the program had trouble with identifying elements and on some occasions the program crashed suddenly. Because of the time limit, the source of these problems was never found and Sahi had to be deselected. Selenium was chosen instead as the tool for record and replay.
Selenium doesn't support AJAX natively, but it provides functionality for the tester to manually make the test case compatible. This functionality is simply to wait until the element is presented to later verify it. Selenium is a free and open source project with a very large community and a lot of beginner tutorials and answered questions.

For the visual recognition strategy, the tools found were Sikuli, RoutineBot and Automa.
All the tested tools had support for testing single-page web applications and were compatible with all browsers. But Sikuli is the only tool that supports other platforms than Windows. Sikuli's core is written in Java, thus can run on any platform that supports Java. Sikuli is also a free and open source tool with large documentation. It was therefore chosen as the tool for the visual recognition strategy.

The supply of testing tools for the random testing strategy was more limited than the others. The other tools and frameworks found were created only for testing specific systems. The only tool found for evaluation was gremlins.js.
Gremlins.js can test single-page applications but cannot detect bugs or generate logs. This tool implements the dumb monkey strategy. It is capable of perform several gestures like click, drag, swipe etc. with different delays on the application under test (AUT).

Ranorex, Silk test and TestComplete implement the keyword driven strategy. All these tools fulfilled the criteria similarly. The Silk test was poor on documentation and it was difficult to find information on the license requirements. So the choice was between Ranorex and TestComplete. TestComplete is a pricy tool with features beyond the scope of this thesis. Since a lot of the documentation on the tool isn't free, the tool was dropped.
Ranorex was chosen instead for its' free documentation and useful tutorials on how to use the tool. Ranorex is only compatible with Windows and has a support for all popular browsers.

For further details on the complete results, see Appendix A.

## 5.2 Strategy criteria

The tools for the different strategies chosen were Selenium, Sikuli, Gremlins.js and Ranorex. These tools were evaluated according to the criteria in 3.2.

It was obvious that the random testing strategy was the only strategy with no support for test oracles and thus unable to test report errors in the AUT. The random testing strategy, gremlins.js, only fails if a crash appears in the AUT. Each test is distinct from the previously occurred ones. This strategy

doesn't need any specifications or knowledge of the SUT thus is very effortless and quick to setup and run, compared to other strategies. It is also useful for stress testing the backend of a web application to check the server response times.

For the handling of graphical user interface (GUI) changes, visual recognition was the only strategy unsuccessful in identifying the different elements when a change in the element's appearance was done e.g. change in color. Sikuli would not be able to locate an element on the screen and failed with the “[error] FindFailed“ exception. Not all GUI changes affects Sikuli though, a change in screen position of an element usually doesn't affect the tests if done correctly. That is by taking a screenshot of the particular element to perform the action on. In this way Sikuli will try to find the element without regard to its surroundings.
Selenium and Ranorex were not affected by GUI changes since they both locate elements based on their IDs or the position hierarchy of the element.
Gremlins.js does not compare any test cases since they are random and unpredictable.

The maintenance of testing tools is a complicated criterion to measure because of the number of different ways a test case can fail.
When using visual recognition, the tester can take a new screenshot of the new changed element if a change in GUI occurs. With record and replay, a tester can either manually modify the step where the test case failed or use the tool to record the new functionality of the AUT.
The maintenance of the keyword driven testing could now be accomplished due to problems that occurred with the Ranorex tool. Maintenance is not required when using the random testing strategy since no predefined test cases are stored.

# 6 Discussion

The keyword driven strategy using the Ranorex tool could not be tested thoroughly since it was not working properly after restarting the tool. Ranorex simply would not recognize certain elements that were present on the screen such as buttons, checkboxes, etc. There was no solution to the bug that was found under the time of this thesis work. Therefor keyword driven strategy was discarded as a consequence of the time limit.

The random testing strategy does not support creating a test oracle for validating a test case, the tests execute at random. It makes it more difficult to find bugs than with other strategies, since the time it takes to find one can vary a lot for each test. Because of the strategy's randomness, bugs can be impossible to reproduce. Hence the random testing strategy is not the best fit for acceptance testing on web applications.

The two remaining strategies are Visual Recognition and Record and Replay. Both of these strategies are able to create test oracles for validating test cases. Both of these strategies are easy to maintain, what separates them is how they handle changes in the graphical user interface (GUI). The two strategies have their advantages and drawbacks in this property, hence they are applicable for the acceptance testing of different web applications.
With Visual Recognition there is a large dependency on the GUI. With this strategy, an element is identified by a screenshot whereas with Record and Replay, an element is identified by its' ID. This was the only significant quality that separated these two strategies. To choose one of these strategies depends on what type of web application is tested and which aspect of it is updated more frequently (the GUI or it's underlying structure). This relation can be seen in figure 6.1.

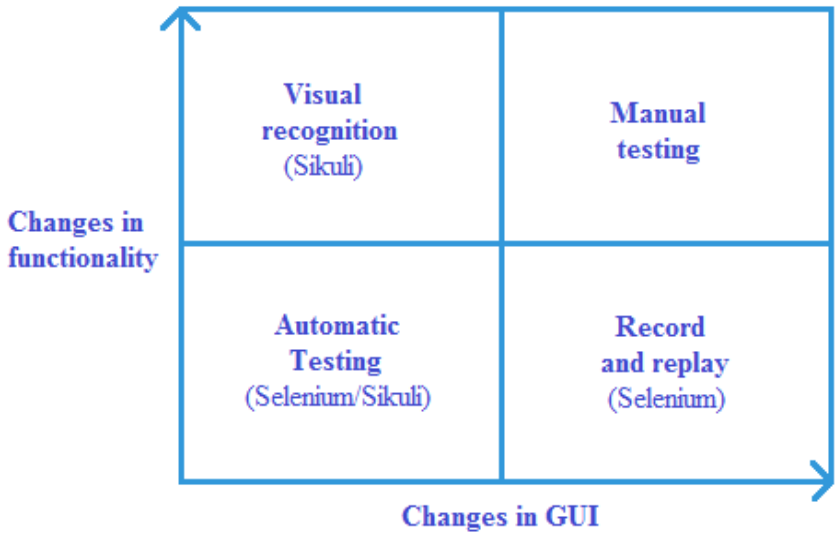


Figure 6.1: Which testing strategy to use in which scenario.

Auddly often modifies and improves their GUI. It is important that test cases don't break as a result of a GUI change. This would cause a lot of time and effort in maintaining their acceptance testing. Therefore a tool with no dependences on GUI changes is required.

With automated tests, more tests can run in a short time in comparison to the manual approach. These tests can also contribute to a better workflow when included in continuous integration. The

developer can run the acceptance tests (AT) immediately when adding new functionality and also get a response directly. These reports can also be shared among the entire developing team.

With manual AT, the application is more perspicuous and the whole application is interpreted at once. In comparison to automated AT's, where the user experience is being missed out on since it only tests defined functionalities. If the tester finds a bug at manual testing, it provides hints on where other bugs can be found.

Automated AT's are most suitable for regression testing. The effort spent on testing old functionality can become repetitive and redundant. It is cheaper to do regression testing automatically, since it saves work hours for the tester.
New functionality is generally more suitable to test manually. It can become unprofitable having to create a new test for each new functionality, since a new functionality is rarely stable at start. Hence it is more suitable to do these tests manually.

# 7 Conclusion

This thesis has contributed with extended knowledge on automated acceptance testing by evaluating and comparing different testing strategies and their tools. Every evaluated strategy performed well in some aspects whereas failed in others. None of the evaluated strategies fulfilled all the criteria for a most effective strategy for testing web applications.

Through the analysis it turned out that with a changing graphical user interface (GUI) the Record and Replay strategy is the most suitable strategy. The Visual Recognition strategy is a better choice if the underlying structure changes more frequently.
Automated acceptance testing is best used for regression testing, when there are minimal changes in both GUI and the underlying structure. Otherwise it is more suitable to test with a manual approach.

Based on the qualities of Auddly's web application, Selenium is the best fit for automated acceptance testing. Selenium maintains a single-page application where the GUI is often modified.

## 7.1 Future works

The limitation of time made it impossible to cover all areas of acceptance testing (AT). During the time of this work, a list of possible future project was developed.

- Examination of the most essential parts of a web application to write test cases/test suites for.
- Using the results from this work to further analyze and evaluate with the model-based testing strategy.
- A calculation of the performance of the different strategies over a period of time.
- A review on the process of deploying a test case for each strategy in a continuous integration server.
- Comparing scripting with using a tool to for the creation of test cases.

# References

[1] A. M. Memon, “GUI Testing: Pitfall and Process”, Computer Vol: 35, Issue: 8, 2002, p. 87-88.

[2] K. Li, M. Wu, “Effective GUI Test Automation”, ISBN: 978-0-7821-5067-4, SYBEX Inc, 2005.

[3] B. Haugset, G. K. Hanssen, “Automated Acceptance Testing: A Literature Review and an Industrial Case Study”, Agile, 2008, p. 27 - 38.

[4] P. Abrahamsson, M. Marchesi, G. Succi, “Extreme Programming and Agile Processes in Software Engineering”, Springer, 2006, p. 331 - 336.

[5] A. Pimenta, “Automated Specification-Based Testing of Graphical User Interfaces”, Engineering Faculty of Porto University, November 2006.

[6] J. M. Arranz Santamaria, “The Single Page Interface Manifesto”, http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php. Published 21 September 2015. Accessed 15 April 2016.

[7] A. Mesbah, “Analysis and Testing of AJAX-based Single-page Web Applications”, http://www.st.ewi.tudelft.nl/~arie/phds/Mesbah.pdf. Published 2009. Accessed 15 April 2016.

[8] M. Wasson, “ASP.NET - Single Page Applications: Build Modern, Responsive Web Apps with ASP.NET”, https://msdn.microsoft.com/en-us/magazine/dn463786.aspx. Published November 2013. Accessed 15 April 2016.

[9] D. J. Richardson, S. L. Aha, T. O. O’Malley, “Specification-based Test

Oracles for Reactive Systems”, International Conference on Software Engineering 1992, p. 105–118.

[10] M. Fewster, D. Graham, “Software Test Automation: Effective Use of Test Execution Tools”, ACM Press, 1999, p. 559.

[11] X. Yang, “Graphic User Interface Modelling and Testing Automation”, Victoria University, May 2011.

[12] H. Cherifi, J. M. Zain, E. El-Qawasmeh , “Digital Information and Communication Technology and Its Applications”, Springer, 2011, p. 33 - 42.

[13] J. Kent, “Test Automation: From Record/Playback to Frameworks”, http://www.simplytesting.com/Downloads/Kent%20-%20From%20Rec-Playback%20To%20FrameworksV1.0.pdf. Published 2005. Accessed 15 April 2016.

[14] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner, “One Evaluation of Model-Based Testing and Its Automation”, ICSE, 2005, p. 392 - 401.

[15] M. A. Mäkinen, “Model Based Approach to Software Testing”, http://lib.tkk.fi/Dipl/2007/urn009573.pdf. Published 22 May 2007. Accessed 15 April 2016.

[16] N. Nyman “In Defense of Monkey Testing”, Microsoft Corporation, http://www.softtest.org/sigs/material/nnyman2.html. Published 2008. Accessed 15 April 2016.

[17] J. Sjöblom, C. Strandberg, “Automatic Regression Testing Using Visual GUI Tools”, http://publications.lib.chalmers.se/records/fulltext/215187/215187.pdf. Published 22 October 2014. Accessed 17 April 2016.

[18] E. Börjesson, R. Feldt, “Automated System Testing Using Visual GUI Testing Tools: a Comparative Study in Industry”, 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, p. 350 - 359.

[19] Rashmi, N. Bajpai, “A Keyword Driven Framework for Testing Web Applications”, International Journal of Advanced Computer Science and Applications, 2012, Vol. 3, No. 3, p. 8 - 14.

# Appendix A

The complete set of the tools and their fulfillment of the criteria.

### A. 1 Random Testing - Test Monkeys

| **Name of tool**<br><br>***Maintained by*** | **Single-page** | **Web browser** | **Environment** | **Licence costs** | **Support** | **Documentation** |
|---|---|---|---|---|---|---|
| gremlins.js<br><br>*Marmelab* | Yes | Chrome, Firefox, Opera, IE8. | JavaScript framework | Open Source, under the MIT Licence. | Latest commit 2016-03-25 | Git readme + examples how to beginner |

### A. 2 Record and Replay

| **Name of tool**<br><br>***Maintained by*** | **Single-page** | **Web browser** | **Environment** | **Licence costs** | **Support** | **Documentation** |
|---|---|---|---|---|---|---|
| Selenium IDE<br><br>*Selenium HQ* | Yes | Firefox plugin for recording.<br><br>Playback: Android, Chrome, Firefox, Htmlunit, Internet Explorer, iPhone, iPad, Opera. | C#, Java, JavaScript, Ruby, Python.<br><br>Windows, OS X, Linux | Open Source. Apache 2.0 license. Free | Latest commit 2016-03-02 | Wiki, API, Forum, big community and support |
| Sahi<br><br>*Tyto* | Yes | All browsers. | Sahi Script<br><br>Windows, OS X, Linux | Free open source.<br><br>$695 Pro. | Last update 2014-11-05 on Open Source.<br><br>Last update 2016-04-11 on Pro. | Tutorials and API. |

### A. 3 Visual Recognition

| **Name of tool**<br><br>***Maintained by*** | **Single-page** | **Web browser** | **Environ-ment** | **Licence costs** | **Support** | **Documentation** |
|---|---|---|---|---|---|---|
| Sikuli<br><br>*Raimund Hocke* | Yes | All browsers | OS X, Windows, Linux.<br><br>Java re-quired. | Free, open source | Latest release 2015-10-07 | Videos, beginners guide etc.<br>Large documenta-tion |
| RoutineBot<br><br>*AKS-Labs* | Yes | All browsers | Windows, scala<br><br>Pascal syntax. | $495 | Latest release 2014-01-24 | Short documenta-tion, hard to find. |
| Automa<br><br>*BugFree Software* | Yes | All browsers | Windows | €389 Basic license | Latest release 2014-10-21 | Full API documen-tation and simple beginner examples |

### A. 4 Keyword Driven

| **Name of tool**<br><br>***Maintained by*** | **Single-page** | **Web browser** | **Environ-ment** | **Licence costs** | **Support** | **Documentation** |
|---|---|---|---|---|---|---|
| Ranorex<br><br>*Ranorex* | Yes | Internet Explorer, Firefox, Chrome, Safari. | Windows | €690 floating license | Latest release 2016-03-24 | API Document, User guide, Code examples etc. |
| Silk test<br><br>*Micro Focus* | Yes | Internet Explorer, Edge, Firefox, Chrome, Safari on iOS, Chrome and Stock on Android. | Windows | Full version license re-quired, no information found about the full ver. | Latest update 2015-11-27 | Poor on user forums, wiki and blogs. |
| TestComplete<br><br>*SmartBear* | Yes | Internet Explorer, Firefox, Chrome, Edge. | Python, VBScript, JScript, DelphiS-cript, C++Script, C#Script<br><br>Windows | € 2134 - 5446 IDE.<br><br>Closed source. | Latest release 2016-03-29 | Documentations and tutorials.<br><br>Offer courses for a fee. |

TRITA 2016:49