Automated testing of a web-based user interface

by

Sandra Kastegård

LIU-IDA/LITH-EX-G--15/038--SE

2015-06-15
Final thesis

Automated testing of a web-based user interface

by

Sandra Kastegård

LIU-IDA/LITH-EX-G--15/038--SE

2015-06-15

Supervisor: Jonas Wallgren
Examiner: Jonas Wallgren
Abstract

Testing is a vital part of software development and test automation is an increasingly common practise. Performing automated testing on web-based applications is more complicated than desktop applications, which is particularly clear when it comes to testing a web based user interface as they are becoming more complex and dynamic. Depending on the goals and needed complexity of the testing, a variety of different frameworks/tools are available to help implementing it.

This thesis investigates how automated testing of a web-based user interface can be implemented. Testing methods and a selection of relevant testing frameworks/tools are presented and evaluated based on given requirements. Out of the selected frameworks/tools, the Selenium WebDriver framework is chosen and used for implementation. The implementation results in automated test cases for regression testing of the functionality of a user interface created by Infor AB.
CONTENTS

Contents

1 Introduction 1
   1.1 Motivation ......................................................... 1
   1.2 Purpose ................................................................. 1
   1.3 Problem definition .................................................. 1

2 Background 2

3 Theory 3
   3.1 Software testing ..................................................... 3
      3.1.1 Testing concepts ............................................... 3
   3.2 Web application testing ............................................. 4
   3.3 Automated testing ................................................... 4
      3.3.1 Automated testing of a web application UI ..................... 4
   3.4 Frameworks and tools ............................................... 5
      3.4.1 Selenium ........................................................... 5
      3.4.2 Sahi ............................................................... 5
      3.4.3 DalekJS ........................................................... 5
      3.4.4 Jasmine ........................................................... 5
      3.4.5 Other frameworks and tools .................................... 6

4 Method 6
   4.1 Pre-study and choosing a framework/tool .......................... 6
      4.1.1 Requirements .................................................... 6
      4.1.2 Comparing and minimizing the alternatives ................. 7
   4.2 Implementation ....................................................... 7
      4.2.1 Structuring the code .......................................... 7
      4.2.2 Designing the test cases ..................................... 7
      4.2.3 Possible improvements of the UI .............................. 8
   4.3 Evaluation of the test cases ....................................... 8

5 Results 8
   5.1 Pre-study and choosing a framework/tool .......................... 8
      5.1.1 Comparing the alternatives .................................. 8
   5.2 Implementation ....................................................... 9

6 Discussion 11
   6.1 Results ................................................................. 11
   6.2 Method ................................................................. 11

7 Conclusions 12
   7.1 Method and framework ............................................... 12
   7.2 Possible improvements of the UI to facilitate testing .......... 12

References 13

Appendix A: Code examples 14
1 Introduction

1.1 Motivation

Testing is a vital part of the software development process and as web applications are becoming increasingly important in our lives it is crucial that they are tested properly. There are many different aspects that can be tested and the priority of testing has often been to try to find bugs and security issues by going through the source code at a low level, testing server and database communication. The user interface (UI) of web applications have previously been on such a basic level that thorough testing have not been a priority. But as web applications are becoming more and more advanced and dynamic, testing the functionality of the web application UI has become more important[1].

One approach to testing the functionality of a UI is to carry out manual testing and have testers using the UI and report any problems they come across, often while following a test plan. The advantage of this approach is that the web application gets tested in regards to the functionality of the application based on the reactions and experiences of an actual user. The disadvantage is that it can be time consuming and expensive[1].

Using automated testing in software development allows for repeatable tests that can be run multiple times by a computer which makes them less costly and time-consuming than manual testing. The use of automated testing of web applications is becoming more common but in regard to testing the functionality of the UI, automation is challenging. Most web applications are dynamic rather than static, which makes them complex to automatically test as the content and elements can change. Web applications are often heterogeneous, meaning that they are comprised of components built using different languages and techniques, which can also make automated testing difficult. The wide variety of web browsers available today is another aspect that makes the automated testing of a web application complicated, since users expect the same performance from a web application regardless of which browser is used[2].

1.2 Purpose

The purpose of this thesis is to investigate how automated testing can be applied to a web-based UI, provided by the company Infor AB, and what established relevant testing frameworks/tools exist that can facilitate this. A few popular frameworks/tools for automated UI testing will be presented and one of them chosen to be used for testing the UI. Potential adjustments, properties and design considerations regarding the UI that could improve the implementation of automated testing will also be discussed.

1.3 Problem definition

- What is the appropriate method for automated functionality testing of the UI and how can it be implemented using a chosen framework/tool?

- What adjustments to the UI are relevant, if any, to facilitate automated testing?


## 2 Background

The thesis work covered in this report is based on an assignment from a team at the company Infor AB that is responsible for an application server for Java applications. The team have automated testing implemented for every part of their work except for their web-based management UI, which means that they have to perform repetitive manual testing of the UI for each release. The UI has recently been re-designed and Infor wishes to receive recommendations regarding an appropriate test framework/tools as well as an introductory implementation of automated tests for the UI.

The UI is part of a web application used by the company to show activity information regarding a grid. In the grid there are different hosts and each host has applications installed on it. The UI consist of a home page that shows a basic overview, including some notifications that consists of alerts, information and warnings (see figure 1). It also has lists of hosts and applications.

At the top of all pages of the UI there is a navigation menu with the following choices: Home, Hosts, Applications, Nodes, Monitoring, Security and Configuration. At the time of this thesis work, the UI was still under development and not all parts had been transferred from the old UI. These parts were not tested, since the focus was on making tests for the new UI. The main parts to be tested was the Home, Hosts, Applications and Monitoring pages. The UI is developed in Bootstrap\(^1\) with AngularJS\(^2\) and did originally not have any ID attributes attached to its elements.

---

\(^1\)https://github.com/twbs/bootstrap

\(^2\)https://angularjs.org/
3 Theory

3.1 Software testing

Testing is used in the software development industry as a quality assurance of the different parts of a software project. It is not only practiced at the finishing stages of development but rather during all stages in different ways. For example there are development strategies based in using tests that specify the requirements and then alter the program until it passes the tests. In other cases testing is used for software that is constantly updated and released in new versions, to check that parts of the software that previously worked still do[1].

Testing is sometimes perceived as demonstrating the absence of errors, when usually testing is a process used for improving the assurance of the reliability and quality of the software by finding as many errors as possible[1].

3.1.1 Testing concepts

There are a wide variety of test categories, techniques and methods out there, focusing on different goals and testing different aspects of software. There are many commonly used concepts and terms regarding testing[3]. This section describes a selection of them.

Unit testing: Unit testing is used to validate individual units in the source code of a system.

Regression testing: When changes has been made to a software, regression testing is used to verify that previously working parts of the software still works. It is especially useful for software that is often updated and released in new versions.

Functional testing: As the name suggests, functional testing is used to verify that a system meets the functional requirements.

End-to-end testing: End-to-end testing is a method for testing whether the flow of an application is performing as designed from start to finish. The purpose of carrying out end-to-end tests is to identify system dependencies and to ensure that the right information is passed between various system components and systems.

Code coverage testing: The idea behind code coverage testing is to measure how much of a system’s code is being tested by a test suite. This is useful for developing better tests that cover more code.

White-box testing: White-box testing means testing the software with regards to its internals. Examples of this are unit testing and code coverage testing. These types of tests are intended for verifying that the system satisfies its functional requirements. White-box testing requires knowledge of the system’s code and design.

Black-box testing: In contrast to white-box testing, black-box testing implies testing the system as a ”black box”, meaning that the internals of the system remain unknown. This kind of testing entails invoking system calls through user interface interaction and correct behavior is only validated by viewing the output of the user interface. Since black-box testing will usually not discover all defects, another kind of testing called gray-box testing should also be applied.
**Gray-box testing:** Gray-box testing is a combination of white-box and black-box testing, which requires partial knowledge of the internal workings and is utilised by the tester in an attempt to discover more defects compared to only using white-box or black-box testing.

### 3.2 Web application testing

Web application testing presents a whole new range of challenges to testing compared to desktop applications. Today’s web applications are often dynamic systems that consists of both a back-end and a front-end and are commonly heterogeneous, meaning that they are comprised of a number of components that are each developed in a different language. Web applications typically support multiple users and operate in a much more open environment than desktop applications do[1].

### 3.3 Automated testing

The concept of automated testing can have different definitions depending on the source and the context. It can refer to the automatic generation of the actual test cases or the automatic running of test cases that have been manually developed. In this report the chosen definition is that automated testing is testing based on manually developed test cases that can be run repeatedly by a script.

There are many reasons for using automated testing. Since the test cases are repeatable and a computer can perform them at high speed, it reduces the time and cost of testing. When used correctly, automated testing can improve the quality of software because a computer can be more exact in registering defects than any human tester. Other positive aspects of using automated testing include increased test coverage, replacing labor-intensive tasks and effective regression testing[3].

#### 3.3.1 Automated testing of a web application UI

As previously stated, testing web applications is challenging, especially when performing automated testing. Functional testing of the UI adds another aspect of complexity, since the UI of dynamic web applications can change depending on the browser used. UI’s can also change in unsuspected ways based on the user activity.

A common approach to testing is the so called capture-replay method (C&R), where test scripts are generated by recording activities performed by a user on a web page. The easy and fast way to generate test cases is a positive aspect of C&R. However, the problem with C&R is that the accuracy of the test is dependent on the location of elements on a web page, which can result in fragile test cases that fail because an element has a slightly altered visual position[4].

Another available method is to program the test cases from scratch and to use different techniques to locate elements by using the source code instead of relying on the visual location. These techniques include locating elements using their HTML ID attribute or by using their XPath, which is a path to exact placement of an element in the source code. Using these techniques means that the visual location of an element is not going to influence the test result. Although this method is better than C&R in that regard, it also implicates a more difficult and time consuming process of creating the test cases. This method most commonly means that the test cases are script-based and needs to be developed manually, often in a programming
3.4 Frameworks and tools

Automated testing frameworks and tools are an increasingly popular area and consists of a wide variety of alternatives. The following sections present a few alternatives that has been considered.

3.4.1 Selenium

Selenium is an open source suite of testing tools which is one of the most established alternatives regarding test automation of web applications. Selenium both have an IDE that can be used for C&R test generation as well as the WebDriver framework that can be used for browser automation and programming test cases. The browser automation used by WebDriver works with most common browsers, such as Chrome, Firefox and Safari. The WebDriver framework supports test cases written in several different languages, including Python, Java, JavaScript and C# [5].

3.4.2 Sahi

Sahi is a tool that’s available in a free open source version as well as a Pro version at a monthly cost. The basis for test automation with Sahi is that test cases can be generated using C&R and then the generated test cases can be built upon using the ”Sahi Script” language (which is an extension of JavaScript). Regarding the web browser control there are Java and Ruby drivers available that works with most popular browsers. Additionally the Pro version includes built-in features for generating reports, storing reports in database, taking snapshots and more. Since Sahi is based on generating scripts automatically using the C&R feature, theoretically the required programming skills are basic[6].

3.4.3 DalekJS

DalekJS is a UI testing tool that uses a browser automation technique, with which the WebDriver JSON-Wire protocol is used to communicate with the browsers. Tests are written in JavaScript. DalekJS is still under development and is not recommended for production use by its creators [7].

3.4.4 Jasmine

Jasmine is a framework for testing JavaScript code. It is behavior-driven, not dependent on other JavaScript frameworks and does not rely on DOM\(^3\) or on browsers which makes it useful

---

\(^3\)A Document Object Model is a tree of available objects on a web page[8]
for different kinds of testing. Jasmine uses a syntax that is created with the purpose to be easily read and understood, so tests are written in such a way that they can be read as sentences[9].

3.4.5 Other frameworks and tools

Besides the selected frameworks and tools presented in the previous sections there are many other alternatives with different uses and specialties. During the research for this thesis the following alternatives were found and ultimately rejected in the first stage for different reasons (see section 5.1.1 for more details): Watir, QUnit, AutoIT, Capybara, Protractor and Robot Framework. Watir is a selection of libraries written in Ruby for browser automation[10], QUnit is a JavaScript unit testing framework[11], AutoIT is a scripting language designed for automating the Windows UI[12], Capybara is a collection of testing libraries written in Ruby that simulate user interaction[13], Protractor is a test framework used for end-to-end testing of AngularJS applications[14] and Robot Framework is a generic test automation framework[15].

4 Method

4.1 Pre-study and choosing a framework/tool

To be able to choose an appropriate framework/tool for automated testing, a pre-study was executed. This pre-study entailed reading about a selection of frameworks/tool existing at the time. To facilitate the decision, a number of requirements was decided upon.

4.1.1 Requirements

The following requirements were decided upon and are listed below according to priority:

1. **Free of charge**
   Since this is a thesis and the resulting implementation is not necessarily going to be used on a large scale by the company, the chosen framework/tool had to be available for free.

2. **Possibility to do black box testing and simulate user behaviour**
   The testing was going to be done according to the black-box methodology, meaning that the functionality of the UI was going to be tested without regard to how the back-end works. The testing had to be able to simulate user behaviour by clicking on elements and filling out input fields and then verify that these actions worked as expected.

3. **Not dependent on other frameworks/tools**
   Many available frameworks and tools are dependent on other frameworks and libraries to work, and might only supply part of the needed functionality. A framework that was as independent as possible and also covered the needed functionality was a requirement.

4. **Ability to develop test cases from scratch**
   For a more thorough control of development and behavior of the test cases, preferably the chosen framework would enable development of test cases from scratch without the help from C&R features or similar techniques.
5. **Ability to write tests in the Java programming language**

The code needed to be easily comprehensible, so that the employees at Infor could eventually continue working on the testing after the thesis was finished. Since most of the employees are used to working with Java, this was the preferred language to develop the tests in.

### 4.1.2 Comparing and minimizing the alternatives

Initially a selection of around 10 different tools and frameworks was researched, all of which are mentioned in the theory section 3.4. This selection was then narrowed down to the subset that are described in more detail in the theory section 3.4.1-3.4.4. This first cut was based on that when examined, none of the discarded alternatives fulfilled more than a couple of the requirements listed in section 4.1.1 or had properties that clearly made them unsuitable alternatives. The final subset was then more thoroughly compared in regard to the requirements, the details of this comparison can be found in section 5.1. The final choice was the Selenium WebDriver framework.

### 4.2 Implementation

#### 4.2.1 Structuring the code

Because the testing was implemented by programming each test case from scratch, a substantial amount of code was going to be written. Since the purpose of the thesis is for the company to continue using the tests and develop them further the code needed to be clear and have a good structure.

The Selenium WebDriver framework can be implemented in many ways, and regarding the structure of the code one of the approaches recommended by the Selenium creators is to use a design pattern called Page Objects[5].

Page Objects is a way to abstract the code by hiding away the actions performed on the different elements used in the testing and separating the test cases from the more "back-end" methods such as finding elements. A “page” can be an actual page of the web application or a part of the application that should be tested as a unit. A variation on this design pattern was used for the implementation.

The code was divided into different pages, each having a separate class containing methods for locating different elements needed for testing that particular page. Then each of these pages had a related testing class, where the actual test cases was written. The test cases utilized the JUnit Eclipse plug-in for the structure and running of the tests. With JUnit it was possible to use @Before, @After and @Test syntax to divide the code in the tests appropriately. This structure can be seen in the code example in appendix A.

#### 4.2.2 Designing the test cases

Each test case was made as atomic as possible and mainly tested one element, the reason being that if one test case failed, no other test cases would be affected. The majority of the testing procedures consisted of verifying that a link was not broken or that a button worked as expected. To accomplish this a link or button was clicked using methods from the WebDriver framework. When the link had been clicked, the new URL and page title was verified as correct.
Assertions was used to confirm that an action worked as expected, by checking that the new URL contains part of the link or that the title of the page is correct.

4.2.3 Possible improvements of the UI

One request from Infor was that any possible changes and improvements to the UI that could facilitate the testing would be reported. The method for discovering these was simply to investigate any problems that would occur during implementation and decide if these could be helped by changes to the UI.

4.3 Evaluation of the test cases

To make sure that the test cases did not fail without reason and that there were no bugs in the code, the tests needed to be run repeatedly. During development the tests were run continuously to make sure that any changes worked as expected. When the implementation was almost finished, a more extensive running of the tests was performed to try and catch any problems that would only manifest sporadically. For this purpose the test cases was run 20 times with the results of each run documented and evaluated.

5 Results

5.1 Pre-study and choosing a framework/tool

5.1.1 Comparing the alternatives

Initially, before any extensive research had been done, the possible alternatives included ten different frameworks/tools. The following alternatives were rejected at an early stage:

- **Watir** does not perform user simulation or black box testing and does not support developing test cases in Java.
- **QUnit** does not support development in Java and is dependent on other frameworks/tools to enable user simulation.
- **AutoIT** is primarily made for automating and testing the Windows GUI, not for testing web-based UI’s.
- **Capybara** does not support development in Java and is mostly used in collaboration with other frameworks/tools.
- **Robot Framework** does not support development in Java and is mostly used in combination with other frameworks/tools.
- **Protractor** is primarily for end-to-end testing, which was not the desired testing method for this thesis.

When the initial alternatives had been minimized to only include Selenium WebDriver, Sahi, DalekJS and Jasmine (listed in more detail in section 3.4.1-3.4.4), a more thorough comparison was conducted based on the requirements listed in section 4.1.2:


- **Selenium WebDriver**
  The Selenium WebDriver framework fulfill all the listed requirements. It is also extensively documented and the WebDriver framework forms the basis for many other testing frameworks/tools.

- **Sahi**
  The free, open-source version of Sahi does fulfill most of the requirements, but it does not allow for tests to be written from scratch or in Java. The Pro version seems more promising but since it is not free it is not a valid alternative for this work.

- **dalekJS**
  DalekJS does fulfill several of the requirements, as it is open source and offers browser automation. But it is still in development and not recommended to be used for production, which excludes it from the list of alternatives.

- **Jasmine**
  Jasmine passed the initial rejection because of it being seemingly established. It does fulfill some of the requirements but seems aimed at a more code focused testing, not for testing web elements. It does not support development in Java and it is mainly used in combination with other frameworks/tools.

When the comparison was made the Selenium WebDriver framework stood out as the clear choice, given that it fulfills all given requirements and is seemingly established when it comes to automated testing of web applications. The extensive available documentation was also a motivation for the choice.

### 5.2 Implementation

In the finished version of the implementation, the code was divided into five different parts, each having their own collection of test cases. These parts all represented different sections of the UI, one part was the top menu visible from all pages in the UI. All the buttons in this menu was tested by having test cases where the menu buttons where clicked and the resulting URL and title was verified. The other four areas represented the Home page, the Hosts page, the Applications page and the Monitoring page. The structure was based on the previously mentioned Page Object design pattern (section 4.2.1) to achieve clear and sufficiently abstracted code.

The five test parts resulted in a total of 27 different test cases. Many of the test cases have a similar structure but perform tests on different elements. Since the implementation is supposed to give an insight in the possible applications of automated testing on the UI, test cases with different uses and techniques was developed to show the range of the framework. Test cases that tests dynamic elements such as a drop down menu and a sorting functionality were made for this purpose. A code example for one of the test cases can be seen in listing 1.
All test cases work independently of each other and are made as atomic as possible. This strategy was used to ensure that test cases would not fail because of the influence of each other. The main part in making the tests atomic was using a new driver for each test case, meaning opening and closing a new window of the browser before and after each test. This made the tests slower, but ensured that the test cases would not cause each other to crash because they were using the same driver. Initially the same driver was used for each suite of tests, this made the tests faster but also meant that a domino effect would arise if one of the tests failed and the driver ended up at the wrong place in the browser automation.

The choice to write the test cases from scratch instead of using other methods such as C&R (see section 3.3.1) was motivated by a wish to develop stable test cases and to have as much control as possible during the development since it is a possible foundation for future development of tests for the UI.

During the implementation two main problems occurred: a timing issue and the lack of ID attributes on HTML elements. The timing issue was not a problem that depended on the UI design, but on the framework. The lack of ID attributes was not a significant problem but resulted in the usage of the less reliable XPath. The XPath expressions works well as long as the elements are not being moved in the source code, in which case they will fail, meaning that they can cause problems when used for testing code that are still in development. A code example for locating an element using an XPath can be seen in listing 2. For a more detailed code example see appendix A.

The finished implementation is a version of black-box testing, appropriate for regression testing. One of the complicated aspects of writing tests using the Selenium WebDriver framework is timing. The driver that controls the browser tries to wait for page loads but it does not always succeed, possibly resulting in the driver not finding an element in time and causing tests to fail because of timing issues instead of an actual functional error of the UI. Since these timing issues would mostly occur randomly, to find these types of problems the test cases was run repeatedly.
When this was done at the finishing stages of implementation (see section 4.3) a small number of test cases failed due to timing issues that resulted in elements not being found (3 out of 27 test cases during 20 runs). This was considered a small problem but still an attempt to fix it was made.

The problem was caused by a localization method failing to locate an element during a given time-frame which caused it to throw an exception. The reason for this was mostly that the page was loaded to slowly. In an attempt to solve this, code for catching exceptions was added to all localization methods so that if the element was not found the first time the method would try once more. This helped and at a second run through all tests passed.

6 Discussion

6.1 Results

The probable result of the research and comparison between frameworks/tools was evident at an early stage of the research as Selenium WebDriver stood out from the beginning as a popular and well established framework/tool for web-based UI testing. Further research solidified this impression.

The research conducted gave the impression that the required black-box testing of a UI could be challenging but the implementation went well without any serious issues. One aspect that did correspond to the facts presented in the theory was the timing issue. This issue is probably a consequence of the dynamic and complex design of today’s web applications discussed in section 3.3.1. A simpler web application would be less likely to cause any delays that would result in timing issues.

6.2 Method

Since the focus was on selecting an appropriate framework/tool and not on any kind of measurement, a high replicability of the method was challenging to achieve. A more detailed method, maybe using some sort of a point system could have made the method more replicable. The current method can be followed step by step even if those steps are pretty broadly defined.

If the method were used with the same or similar selection of alternatives and with the exact same requirements you would get the same result, giving the method some reliability. But change any of these aspects and the result would differ. Some of the deciding factors are subjective, making the method less reliable.

The evaluation of the UI was continuously done during the implementation with possible suggested improvements based on solutions to any problems that appeared. With more time available, the method for the evaluation could have been defined in a more elaborate way and include doing research on the techniques used to develop the UI.

Had there been more time, a wider selection of alternatives could have been researched. The alternatives could also have been researched even more thoroughly and a last remaining small subset of alternatives (like those mentioned in section 3.4.3-3.4.4) could have been evaluated by doing small-scale implementations with them.

Some considerations were made concerning the sources used. Since the area of software testing and especially web application testing is ever changing and many of the frameworks/tools only have been developed during recent years, the sources needed to be current. No sources
older than 10 years were used because of this, which resulted in a smaller selection of available sources but hopefully made the findings more relevant. Extra effort was made to find valid sources for the theory section that was not blogs or forums. The testing community is very active on-line and this was very helpful for the implementation, but for the theory research the focus was on using more scientifically valid sources.

7 Conclusions

The main purpose of this thesis was to investigate how automated testing can be applied to a web-based UI and what available testing frameworks and tools could be used to facilitate this. The purpose was fulfilled and a selection of possible test frameworks/tools was presented. The implementation resulted in test cases developed for regression testing which can be used as a foundation for further development by the company which was also part of the purpose.

7.1 Method and framework

For automated testing of the functionality of the UI a black box testing method was deemed the appropriate choice. The testing was implemented in a satisfactory way using the Selenium WebDriver framework for browser automation and for performing user simulation. Small and atomic test cases that used assertions for testing was created. The testing is focused on the functionality of different elements in the UI which was the purpose and part of the problem definitions.

7.2 Possible improvements of the UI to facilitate testing

During and after the implementation of the testing one main possible improvement to the UI appeared: adding IDs to the elements. While using XPath expressions worked for the implementation, XPath expressions are in general less stable for code still under development, since they are based on the position of the element in the source code. Moving an element in the code causes the tests to fail due to not being able to find the element. Using IDs for localization means a more durable test code that are less likely to fail due to elements changing position. Besides this aspect there were not any prominent issues with the UI that emerged during the implementation.
References


Appendix A: Code examples

Code example that shows one test case for testing the Home page as well as the JUnit syntax with @Before, @After and @Test.

```java
public class HomePageTests {
    /** Firefox Browser driver**/
    private static WebDriver driver;
    private final static String HOST_PORT = "https://";

    /** Before-method that is run before each test case. **/
    @Before
    public void before() {
        driver = new FirefoxDriver();
        driver.get(HOST_PORT);
    }

    /** After-method that is run after each test-case. Quits the driver. **/
    @After
    public void after() {
        driver.quit();
    }

    /** Test case that checks that the "Hosts" link on the home page works */
    @Test
    public void testHostsNameLink() {
        System.out.println("−−−−−−−−−−−−−Hosts link test START");
        HomePage.hostsNameLink(driver).click();
        String curURL = driver.getCurrentUrl();
        assertTrue("Current URL contains #/hosts", curURL.contains("#/hosts"));
        System.out.println("−−−−−−−−−−−−−Hosts link test END");
    }
}
```
Code example containing a method for finding an element. This method is used by a test case for when testing the Home page.

```java
public class HomePage {
    private static WebElement element = null;

    /** Locates and returns the "Hosts" link from the Home page **/
    public static WebElement hostsNameLink(WebDriver driver) {
        String elemPath = "html/body/div[3]/div/table/tbody/tr[1]/td[2]/div/a";
        try {
            element = (new WebDriverWait(driver, 30)).until(
                        ExpectedConditions.presenceOfElementLocated(By.xpath(elemPath)));
        } catch (Exception e) {
            element = (new WebDriverWait(driver, 30)).until(
                        ExpectedConditions.presenceOfElementLocated(By.xpath(elemPath)));
        }
        return element;
    }
}
```
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära 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 upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman 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 upphovsmannens litterära eller konstnärliga anseende eller egenart.

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 owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her 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/

© Sandra Kastegård