Phat Chau Tan

Automation Testing With Robot Framework

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

27 March 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Phat Chau Tan Automation Testing With Robot Framework 36 pages 27 March 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Jaana Holvikivi, Principal Lecturer

Software testing is now a key part of the software development process. Especially, automation testing is more and more required by companies. The goal of the project was to use Robot Framework to build tests that perform automation testing of an application's user interface.

The project used Selenium2Library, Python version 2.7.11 and Robot Framework version 3.0 in a Windows environment. Robot Framework is free of charge and a powerful tool for automating testing activities. It has various testing libraries and a supporting community. Test files are organized in Pycharm IDE. Selenium2Library is to import keywords to perform user interface testing. The report demonstrates the implementation of test scripts and testing environment setup.

Two test suits were created for login and item search testing. Robot Framework generates clear and simple reports so that a user can easily locate specific information. The project also brings experience and learned lessons about how to write better testing scripts. Automated tests help reduce time and the cost of running test cases and fixing the application's bugs.

Keywords	Automation, testing, Robot Framework, Selenium2 Library, Python

Contents

Abbreviations

API	Application Programming Interface
ATDD	Acceptance Test-Driven Development
CMD	Command Prompt
CSS	Cascading Style Sheets
DDD	Data Driven Development
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IE	Internet Explorer
UAT	User Acceptance Testing
URL	Uniform Resource Locator

1 Introduction

Competition in the Information Technology field becomes higher and more challenging so companies and start-ups have to strive for new ideas and methods to survive. Hence, management and developing teams need to deliver quality products that meet custom needs. Building and maintaining quality software is a challenging task since customers often change their requirements and projects get larger and more complicated. Therefore, software testing is a compulsory phase of current projects to ensure the high quality of end products.

Automation software testing is a necessary and important phase in the current software development process. Testing activities are to discover errors and defects of applications in an early phase to produce reliable products to be released in production. To accomplish this, the quality assurance team has to perform proper manual or automation testing. In this report, I will investigate in detail how automation testing works.

The goal of this project is to use Robot Framework to develop automation scripts using Selenium2Library to test graphical user interface (GUI). I will investigate more to see how users can customize their automated scripts and how Robot Framework performs automation testing. The learned lessons and experience drawn from the project are also discussed in this report.

2 Testing in Software Development Methodologies

2.1 Waterfall

Waterfall is one of the popular software development models in the world. This development method is linear and sequential which means that each development phase must be completed before moving to the next phase. Typically, the Waterfall model has the following stages: System and Software Requirements, Analysis, Program Design, Coding, Testing and Operations including deployment and maintenance.

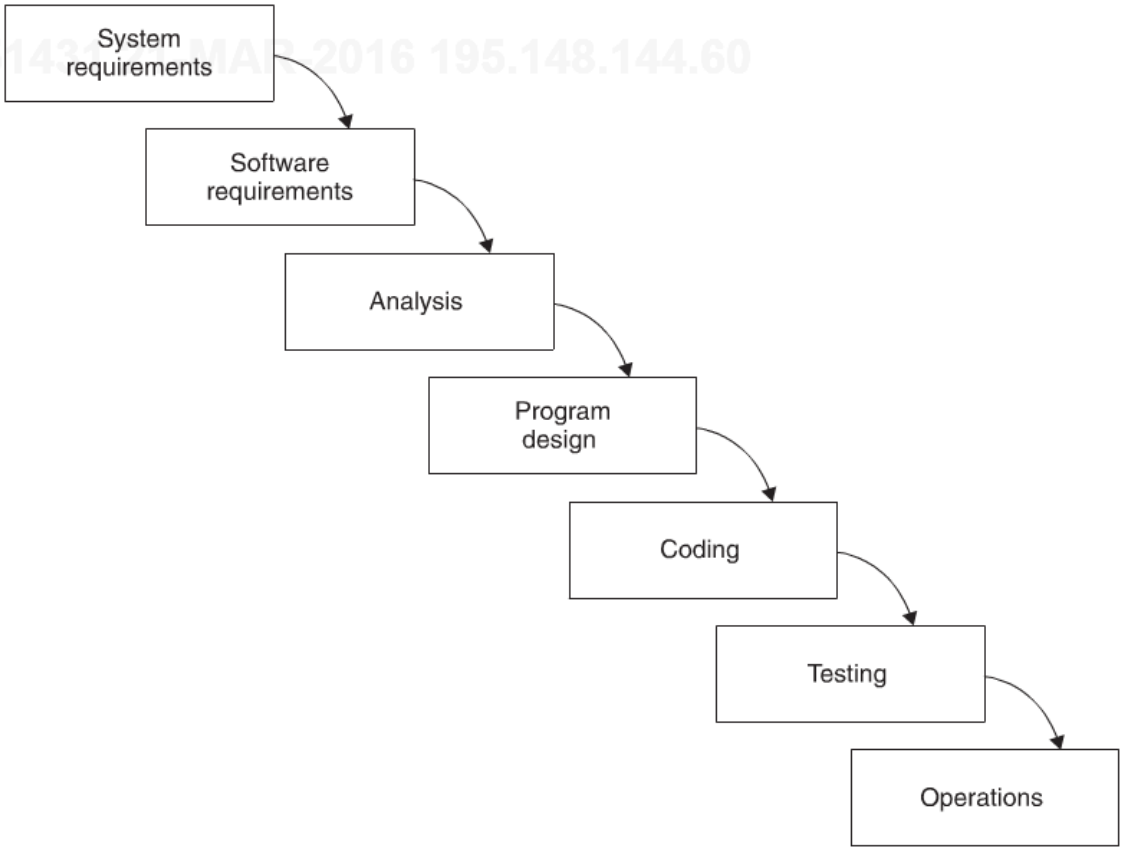


Figure 1. Waterfall Model [1].

Figure 1 represents the phases of the Waterfall development methodology. Testing activities can be started early by performing reviews of specification documents after all system and application specifications are completed. A document review is a necessary step in software testing to eliminate unnecessary documentation bugs and to ensure that all requirements are well clarified before the testing phase begins. During the verification phase, if clients want to change or update requirements, customers have to wait until the test round is completed because documents cannot be updated.

When testers confirm that the software works well according to client specifications, developers will release the software in production. Since then, the project moves into the maintenance phase. This is the longest phase of the whole project. At this stage, production bugs or the customer's new update requests will trigger the whole process again to address the customer's demands. One major drawback of this methodology is that since the testing phase is performed at the end of the process, bugs and errors are found at a late stage of development process. It is more expensive and difficult to fix defects at a later stage than an earlier one.

2.2 V-Model

V-Model is another sequential development process. This is an extended version of the Waterfall model. V-Model stands for Verification and Validation, which means that every development phase has an associated specific testing phase. Verification means testing activities. Similar to the Waterfall model, the next stage begins only when the previous stage is totally completed. [2.]

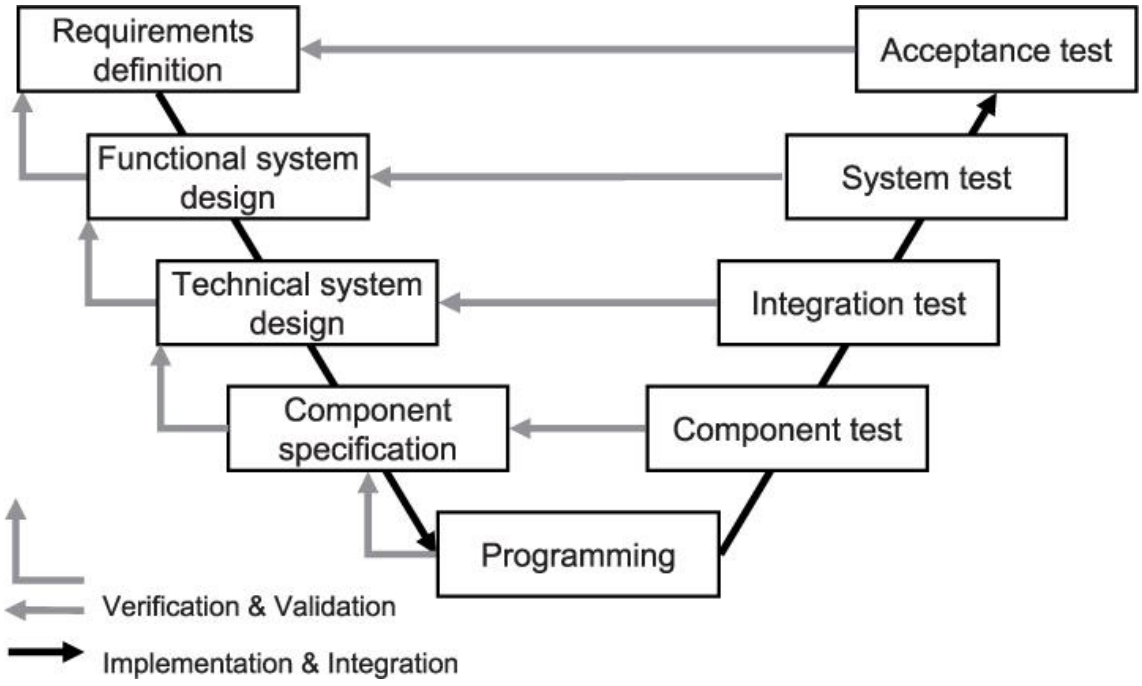


Figure 2. V-Model [2].

Figure 2 illustrates that software development and testing are performed simultaneously. Testing actions start right in the Requirement Definition phase. The fact that testing

activities begin at an early stage helps identifying bugs and defects earlier. This will reduce the cost of fixing bugs.

The left side of Figure 2 represents the development phases that each of them has associated test planning process to prepare for actual test execution stages in the right side of the figure. In the Requirement Definition phase, when the development team understands clearly all client's requirements, testers or verification specialists will review and use these specifications to create test inputs for User Acceptance Testing (UAT) test cases. Next phase is Functional System Design where after the team identifies product requirements, a complete system design is created to describe how the requirements can be implemented. Testers again will review the whole design and develop a system test plan. Creating a system test plan in this stage saves quite amount of time for actual execution of testing in later phase. Next phase is Technical System Design stage that more detailed and technical matters are discussed and planed by developers. In associated with this phase, integration test plan is created. In the last phase Component Specification, each and every component will be determined on how each of them is implemented. Based on design details of each module, a unit test plan is created. [2.]

2.3 Scrum

Scrum is a framework developed to manage from simple to complicated Agile projects. It is not only used in software development but also in other areas. The primary purpose of the Scrum methodology is to allow a team to respond quickly to changes by focusing on prioritized tasks. [3.]

Figure 3. Scrum process [4].

Figure 3 demonstrates a typical Scrum process. A product backlog contains a list of software features, updates and tasks to be implemented. The development team will go through the sprint planning phase to select tasks for the sprint backlog. Once the sprint backlog is ready, the team will carry out a serial of sprints or increments each of which usually lasts for 2-4 weeks. The goal of sprint is to emphasize the shippable working product at the end of each sprint.

In Scrum, there is no actual phase called “Testing”. Quality assurance activities occur any time during the project. Testing can be considered a separate task in the sprint backlog or as the “Done” criteria for a task. Basically, a Scrum team is cross-functional. For that reason, a Scrum team may not have a dedicated test manager. The Scrum Master or developer can even help organizing and building tests. However, proper testing requires professional skills and experience so the team should have qualified testers. They will be dedicated to managing, creating and executing test cases. [3.]

After every sprint, the product gradually becomes larger and more complicated, so regression and continuous integration testing are required to ensure that the end product still works correctly. Testing all new and currently existing features manually is not the best option while automation testing can do the job faster. In automation testing, initially testers have to spend an amount of time and effort to build automated tests but it will save enormous time when executing the actual tests later on.

Working software is the primary measure of each increment’s process. Nevertheless, testing can also be considered one main measure to assess a project’s progress because the final product cannot be released until development and all associated tests are passed. Testing is a crucial section in a complicated Scrum project. [5,7.]

3 Fundamental Software Testing Processes

3.1 Planning and Control

The first issue in test planning is to understand the purposes of the project and clients, and the potential risks. Once the testing team gains this kind of knowledge about the project, the next step is to determine the goals and specifications of testing activities. However, to facilitate the making of a test plan, the company should have the test policies and test strategy of the company in advance. The test policy is a high-level docu-

ment specifying regulations for testing. The test strategy is a company-level document to define the testing approach to achieve specified testing goals. [6.]

Testing planning has some main activities and tasks. Firstly, it is important to identify the goals of testing and the scope. The scope includes selection of the software, system and components. Risk assessments are also created to point out difficult areas of the project and product, so that testers will address them properly. Secondly, the task is to define the testing approach to describe how the team performs testing. The team will select the testing techniques to be used, areas to be tested and the way that related teams and individuals work and communicate in specified procedures. Thirdly, the plan should determine test resources including the required hardware and software and human recourses. Fourthly, if the company has a test policy and test strategy, the plan needs to follow these high-level documents strictly. The fifth activity is to construct the timeline for all activities such as test analysis, test design and execution, and assessment. Finally, clarification of the exit criteria needs to be made. This gives a definition of how testing activities are to be considered completed properly. Some examples of the exit criteria are 100% requirement coverage, all test cases executed or all critical bugs fixed. [6;7.]

Planning is not enough. It is required to have a test control to monitor testing activities in order to propose any necessary changes during testing. Testers should evaluate the actual progress against the planned progress. Based on information gathered from control activities, testers report to the manager and client about the current testing status and receive any comments to update initial test planning. The test control has several main activities. Firstly, the test control is to measure the testing outcome. The team should know the number of passed and failed test cases. The amount, severity and type of bugs need to be tracked. The second activity is to keep track of testing status, percentage of requirements covered and exit criteria. These should be documented and transferred to the related team and individuals so that the whole team knows the current status of testing and test results. Moreover, the latest testing information should be available to the project manager and key project members so that they can make a decision whether to continue or stop testing. Based on the provided testing data, they can decide whether testers carry on or stop testing, or release the product. The final activity is about taking corrective actions. Testers can send software back to developers and ask for more investigation if they find critical bugs that prevent them from per-

forming further testing. In short, testing control is a continuous activity throughout the whole project. [6.]

3.2 Analysis and Design

While the team creates test planning and test control in planning and control, testers will examine the testing goals further and convert them into test conditions at the analysis and design stage. At this stage, the first testers have to look at the product's requirements and specifications for detailed information to understand how the software behaves. Then they generate test conditions based on the software's specifications. Test conditions describe the objectives of the specifications. An example of a test condition is "When users enter a wrong password more than 3 times, a notification will appear". Moreover, during this phase, testers also discover the uncertainty in the requirements, and figure out the boundary points and values where the software is easy to fail. Unclear specifications will lead to an issue that related requirements cannot be tested. One reason for this issue is that the requirements are very general; hence they must be specific and detailed enough so testers can carry out testing. For example, this is a requirement that does not meet the testing purpose: "The website automatically logs out if there is no user activity". From the tester's point of view, this is difficult to test because they will ask themselves how long the website should wait for a user's response before logging out. Testers will ask project managers or developers to clarify the documented requirements in a more specific way. A more testing-friendly specification is: "The website automatically logs out if there is no user activity in 10 minutes". Finally, testers evaluate what is needed to build an environment for testing. They consider the necessary hardware and software tools such as selecting the automation testing tool, database server, computers or licenses.

3.3 Implementation and Execution

Once test conditions are ready, testers will generate test scenarios and test cases. Test scenarios are a group of test cases that test the same software's requirements. Test cases are specific tests with a clarified input (status before test) and expected output (expected status after test). The following example illustrates the differences between test conditions, test scenarios and test cases. [8,24-25.]

- Test condition: When users enter a wrong password more than 3 times, a notification will appear.
- Test scenarios:

1. When users enter a wrong password more than 3 times, a notification will appear.
2. When users enter a wrong password less than or equal to 3 times, a notification will appear.

- Test cases for scenario number 1:
  1. When users enter wrong password 4 times, a notification will appear.
  2. When users enter wrong password 5 times, a notification will appear.
- Test cases for scenarios number 2:
  1. When users enter wrong password 3 times, a notification will not appear.
  2. When users enter wrong password 2 times, a notification will not appear.

In this phase, test suites are constructed to help organize the test structure. A test suite contains many test cases indicating certain functions of software. Testers also consider a set of different test inputs used in test execution. Eventually, a test environment that is identical to the production one is built. Pre-testing activities need to be done to acquire a sufficient number of computers and devices, and install the required software and tools to prepare for implementing and running tests.

The next phase is test execution when testers run all test cases to fully verify the system under test. The process is started manually or automatically if an automated tool is utilized. Once the test execution is finished, testers investigate the outcome report to see where the tests fail. Failed tests mean that the software does not generate results as expected. Finally, bugs are opened for failed tests in a bug management and control system such as Bugzilla or Mantis.

3.4 Result Evaluation and Reporting

While the first phase of the testing process maps out the exit criteria, the goal of this phase is to check the testing results against pre-defined exit criteria. Testers evaluate whether they have performed enough testing or not. They also consider whether they should do more testing or update the exit criteria as part of control activities. While evaluating the results, testers inspect the test evidence, note what has been and has not been verified and document which area testers have to pay more attention to in the next rounds. The status of bugs also needs to be monitored to see the total number of bugs and how many of them are verified or remain open. Finally, the test round report

must be generated so that all related team members can see the result and what has happened. The data in the result document must match the one stored in the bug management and control system. [6.]

3.5 Test Finalization

The testing team will finalize the testing when the product is distributed to customers or the project reaches a certain target. The project officially moves into the maintenance phase, after the software is released to the client. In this phase, the team gathers figures and facts to assess and improve performance of testing and withdraw a lesson and experience for later projects. Test assets should be saved so that when regression testing or verification of a new update is required, testers can retrieve these assets quickly and utilize them again. Bugs that has not been verified yet need to be documented so next testers will verify them when next round starts.

4 Software Testing Levels

4.1 Unit Testing

Unit testing is designed to find bugs and errors in tiny and independent parts of the code. Software is divided into smaller components, modules and parts. These pieces are separated from the whole software and tested to check their function. Then, they are connected to check whether these components or units communicate with each other correctly. The testing targets of unit testing can be functions, classes or simply a functional part of code. Either testers or developers can write code for unit testing. This is the initial level of testing. To help perform unit testing, drivers and stub are created. The driver acts as a component to call the other target component that needs to be tested. The stub acts as a component to be called from the component to be tested. [9,148-150.]

Doing unit testing brings many advantages to the project. Errors are found at a very early stage of the project; hence the cost of fixing bugs is largely reduced compared to the ones found in the production environment. It is easier to spot and fix bugs in a single component than in a whole program. The tests can also be automated. When testers need to perform regression testing and the project becomes larger with many individual modules, automation will save plenty of time testing all current existing modules. This will greatly assist the team to deliver a higher quality program.

4.2 Integration Testing

Integration testing is carried out after unit testing with the purpose of verifying function, interaction and communication between individual modules. In this phase, small modules tested in unit testing are connected and combined into a larger and more complex component. A dedicated team usually performs the testing. These modules are tested to check whether they communicate correctly after they are integrated. In other words, these modules are tested as a group. Several testing methods are available to be applied in this phase. Two most common techniques are top-down and bottom-up techniques. [9,152-153.]

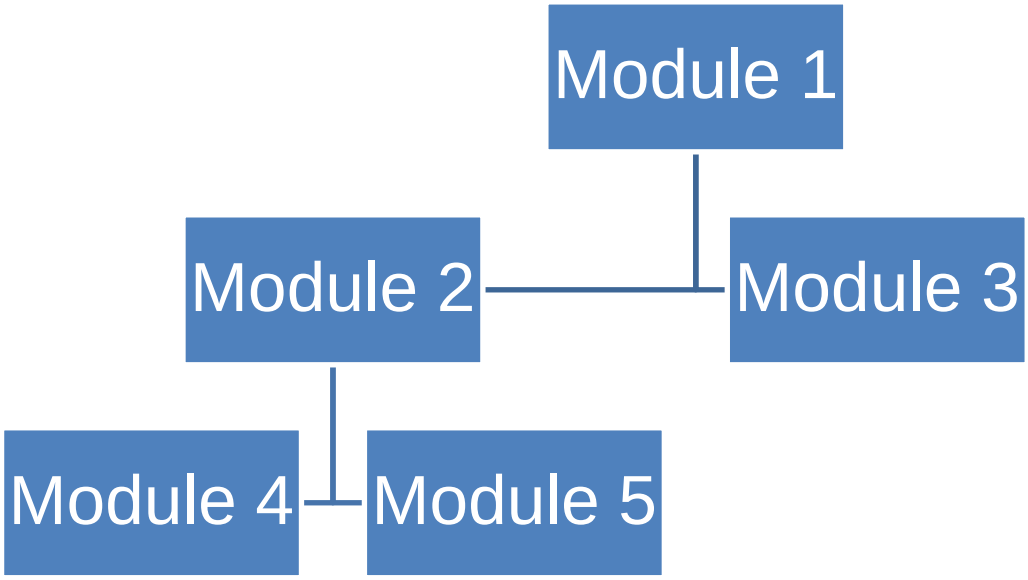


Figure 4. Hierarchy of integrated modules.

Figure 4 illustrates the levels of the integrated modules. The level of hierarchy goes in a descending order from Module 1 which has the highest level and is more critical. The top modules define the navigation and logic flow of the software while the bottom modules represent basic functionalities.

The top-down technique focuses on testing from the top module first to bottom ones. One advantage of this method is that testers usually do not need many drivers. Moreover, an initial sample of complete software may be ready for testing earlier and testers have more time to find severe design bugs at an early stage. However, one disadvantage is that the testing team will need to spend initial resources to write stubs for

testing. Another disadvantage is that verifying the basic functionalities is carried out at the end of the process because testing the bottom modules is performed at a later stage. [9,155-157.]

In contrast to the top-down method, the bottom-up approach starts from the bottom modules that have the lowest hierarchy. A drawback of this technique is that drivers need to be created before testing and implementing drivers is complicated than stubs. Critical modules that are on top of the software architecture are tested at the end of the process and the team may not have enough time to perform thorough testing to search for critical defects. Nevertheless, as testing happens for bottom level modules first, the basic functionalities of software are ensured. The approach brings a time-efficient advantage because testing begins as soon as developers complete basic modules and integrate them. [9,154-155.]

4.3 System Testing

System testing is the last phase performed by dedicated testers to see if the end product meets the customer's needs before handing the software to clients. Testers will carry out full system testing after all components are integrated into a final complete product. In this phase, testers need to act as real users to simulate all scenarios that may happen in a production environment. In addition to functional testing, this phase should include a non-functional check of software. The screen layout and usability need to be inspected and testers can give comments and suggestions to developers to enhance a user's experience. The testing of this phase is based on software specification documentation; hence testers also need to check the requirement document to see whether all requirements are specified correctly. The quality specification document will improve the testing process, so performing requirement reviews is a good practice if timeline allows. In short, this phase is the final gate to ensure that the software meets the client's requirements.

4.4 Acceptance Testing

User acceptance testing (UAT) takes place after system testing has been completed to verify all the software's behaviours. The primary goal of acceptance tests is to check if the software works as expected in real-life scenarios. People who carry out the testing are customers or end users. There are two types of UAT: Internal UAT and external UAT. People from the company that build the product will do the internal UAT first.

These people must not be the ones that directly develop the software but they should be from another department. Then, real end-users will perform the external UAT after the internal UAT is passed. If customers find any valid bugs or make a change request, the software will be sent back to developers to implement and testers will do another regression testing round to verify the changes. Clients will then decide whether they want another round of UAT or not. Once clients confirm the pass for UAT, the project team will finalize the last steps of documentation and the software is released to customers.

5 Robot Framework

Robot Framework is defined as: “Robot Framework is a Python-based, extensible keyword-driven test automation framework for end-to-end acceptance testing and acceptance-test-driven development (ATDD)” [10]. Keywords in the library are written either in Python or in Java. The framework is free to use and published in compliance with Apache License 2.0. The Robot Framework community is very supportive at answering questions and provide quick fixes for issues. It uses a keyword-driven testing method meaning that keywords act as programming functions or methods so it is easy for users to create their own higher-level keywords based on existing ones provided by available libraries. The text syntax follows a tabular style which makes writing test cases more user-friendly. The framework is utilized to perform system testing, acceptance testing and regression testing. The software supports a high-level structure of tests and provides multiple test editors such as RIDE or Eclipse plugin so that users can easily maintain and scale the tests. [10.]

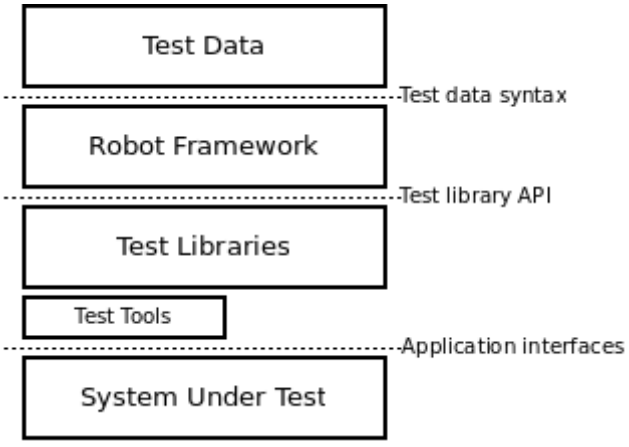


Figure 5. Robot Framework Architecture [10].

Figure 5 illustrates the high level architecture of Robot Framework. It receives the test data and uses test libraries to communicate with the system that is being tested. Interaction between test libraries and the system under test is usually direct. However, some test libraries need to have drivers such as Selenium2Library in order to connect with the tested system. Robot Framework can work with multiple browsers such as Internet Explorer, Firefox and Chrome.

The framework contains many standard and external libraries to support various kinds of testing. Each library serves a unique testing purpose. Standard test libraries are included while installing Robot Framework such as Builtin, OperatingSystem, Dialogs and Remote. On the other hand, external libraries are created to meet the user's desires and requirements to perform certain testing purposes. One great advantage of the tool is that new core keywords can be written in Java or Python to do certain activities of testing. This enhances the testing capabilities of the tool. Some of the most common external libraries are Selenium2Library, Database Library or Android Library.

6 Robot Framework Installation

The project is carried out in a Windows environment. In this project, I use Python version 2.7.11 and Robot Framework version 3.0 which are the latest applicable versions at the time. Actually a newer Python version 3 is available but this version so far has some drawbacks and does not fully support all functionalities.

Prior to installing Robot Framework, Python needs to be installed at https://www.python.org/downloads/. Next the Path variable value needs to be set up for Python as shown in Figure 6 below.

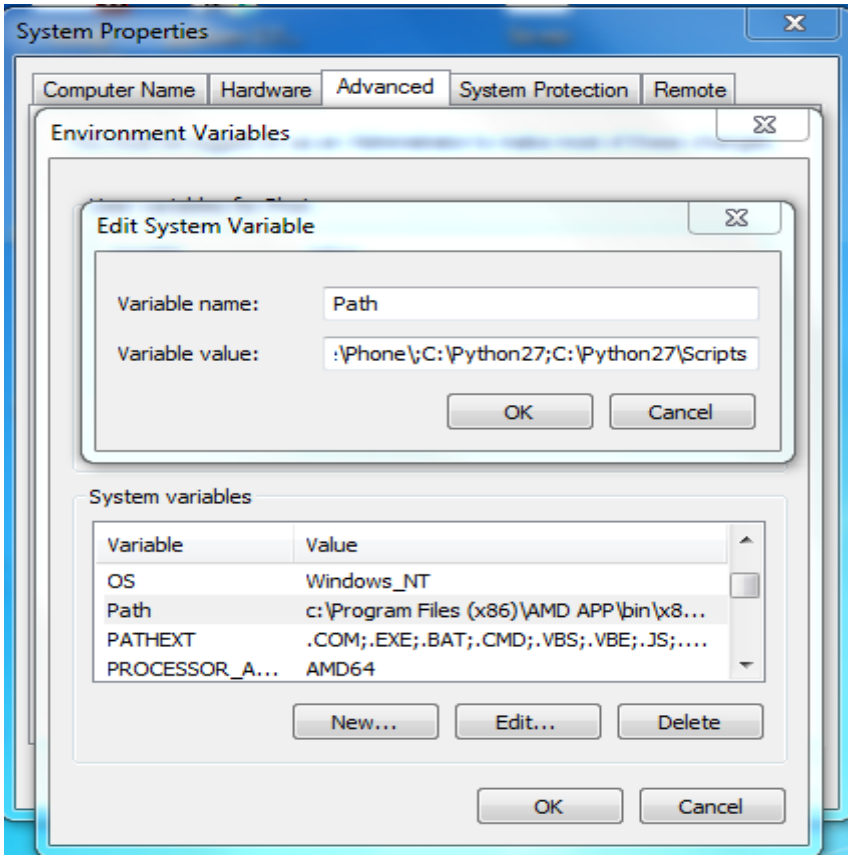


Figure 6. Path variable values.

Figure 6 displays the visual location to update the Path variable value. Instructions for setting up Path value are given below:

- Go to directory “Control Panel\System and Security\System”.
- Select “Advanced system settings”.
- “System properties” popup appears and select “Environment variables” button.
- “Environment variables” popup appears.
- Find the Path variable in “System variables” section and Click “Edit” button.
- “Edit system variable” popup now appears and the value line “;C:\Python27;C:\Python27\Scripts” needs to be added to “Variable value” field.

The next step is to verify whether Python is installed correctly. I open Command Prompt (CMD) interface and type `“python --version”` and it will result in the installed Python version as shown in Figure 7 below.

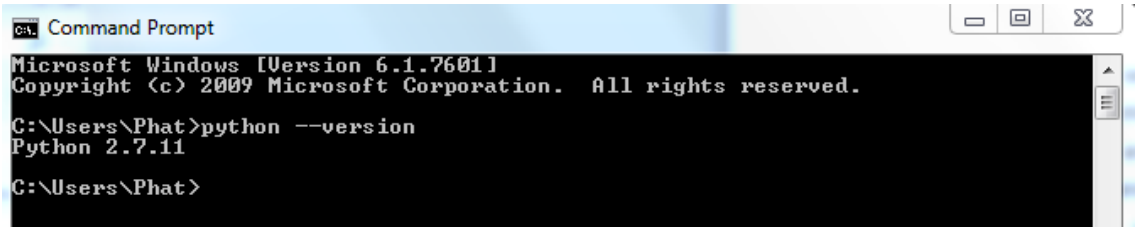


Figure 7. Python version.

Figure 7 shows the installed Python version after the command is entered. Once Python is ready, Robot Framework can be installed by using pip. Pip is a package manager for packages and software written in Python. It is very practical and easy to use. If Python version 2.7.9 or newer versions are used, pip is automatically installed already.

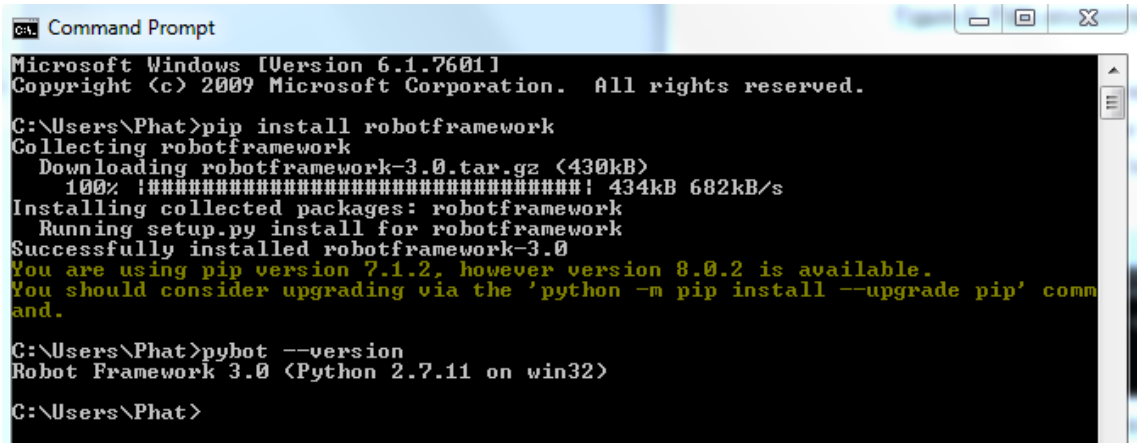


Figure 8. Robot Framework installation and version.

Figure 8 illustrates the command lines to install and verify the Robot Framework version. To install Robot Framework, I open the CMD again and enter "`pip install robotframe work`" [11]. I then once again need to verify that Robot Framework is installed correctly by typing "`pybot --version`". The result line "`Robot Framework 3.0 (Python 2.7.11 on win32)`" shows the installed Robot Framework version 3.0 along with the Python version 2.7.11. Then, I will install Selenium2Library by entering command in the CMD "`pip install robotframework-selenium2library`".

Additionally, the framework has many supported test editors and an integrated development environment (IDE) with plugin to support the management and organization of the test structure. This project uses PyCharm IDE plus Robot plugin. PyCharm is easy to use and brings great experience. It is free for Community Edition and can be ac-

cessed at https://www.jetbrains.com/pycharm/. When I finish installing PyCharm, I need to install a Robot plugin. In the PyCharm interface, I select “File” and then “Settings”. When “Settings” popup appears, I choose “Plugins” in the left panel and select “Browse repositories” button. “Browse Repositories” popup appears and in the search tool, I search for Intellibot and install it.

7 Test Cases Implementation

7.1 Login Test Cases

The Selenium library is one of most commonly used libraries for testing a web application interface. It interacts with the web application through its own driver. Each browser requires different Selenium drivers. Chrome and Internet Explorer (IE) need to have separate drivers; meanwhile Firefox does not require one. The addresses for downloading the latest driver versions for Chrome and IE are given below:

- IE: http://selenium-release.storage.googleapis.com/index.html
- Chrome: http://chromedriver.storage.googleapis.com/index.html

The next step is selecting the latest version of drivers in Win 32 format. Then, the Path variable needs to be configured to point to the folder that has these drivers. For example, if the folder path containing both drivers is C:\Drivers, then ";C:\Drivers" must be added to the Path variable.

It is important to have a good test structure to easily maintain and scale the tests as more and more test cases will be added later on. Test cases for user login and item search are implemented and organized in five files "*Common_Keywords.robot*", "*Common_Login.robot*", "*Common_Login.robot*", "*Login.robot*" and "*Search_Item.robot*". PyCharm editor utilizes the "*robot*" extension format as Figure 9 shows below.

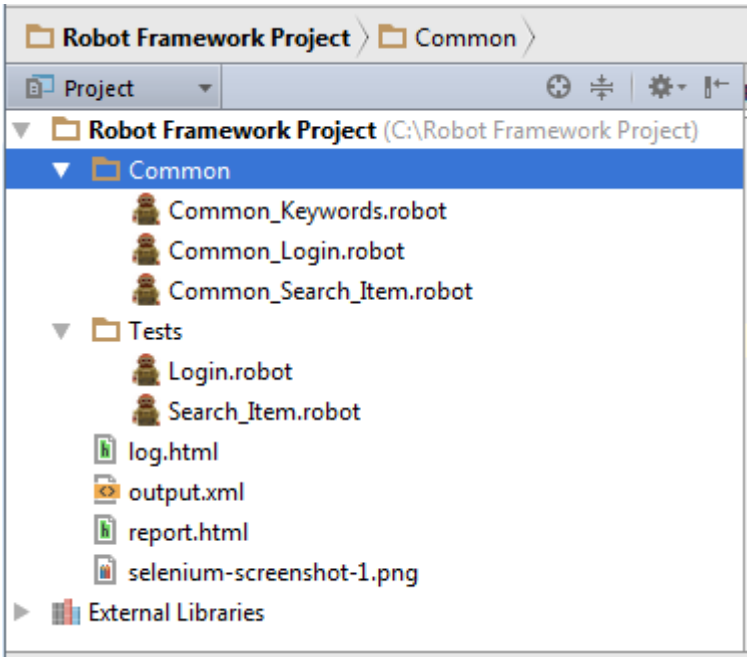


Figure 9. Tests organization in PyCharm.

Figure 9 represents the automated tests structure where "Robot Framework Project" is both a root folder and project name. Inside the project folder, there are two sub-folders "Common" and "Tests". The "Common" folder contains robot files that serve as resource files. These resource files list common keywords that are called many times from different files in the "Test" folder. Hence, it reduces the complexity and size of the execution files in the "Test" folder. The "Tests" folder includes test files that have actual test cases which will be run by the program. "Login.robot" file imports "Common_Keywords.robot" and "Common_Login.robot"; and "Search_Item.robot" imports "Common_Keywords.robot" and "Common_Search_Item.robot". Listing 10 below lists the content of "Login.robot" that users with invalid credentials cannot log in.

```
*** Settings ***
Resource        ../Common/Common_Keywords.robot
Resource        ../Common/Common_Login.robot
Test Setup  Open Browser Test Setup  http://www.sandbox.ebay.com
Test Teardown   Close Window

*** Test Cases ***
Wrong Username And Password Are Entered
    [Documentation]  Test that users cannot login with wrong
username and password
    [Tags]   Wrong_Credentials
    Enter Login Page
    Enter Username          user1
    Enter Password          1234
    Sign In
    Check That Login Fails

Empty Username Is Entered
    [Documentation]  Test that users cannot login with empty
username
    [Tags]   Empty_Username
    Enter Login Page
    Enter Username          ${empty}
    Enter Password          1234
    Sign In
    Check That Login Fails

Empty Password Is Entered
    [Documentation]  Test that users cannot login with empty
password
    [Tags]   Empty_Password
    Enter Login Page
    Enter Username          user1
    Enter Password          ${empty}
    Sign In
    Check That Login Fails
```

Listing 10. Login.robot script.

Listing 10 presents the structure of the Robot Framework file. In the script, each section is separated by "*** ***" line. "*** Settings ***" defines resource files and used libraries and specifies activities prior to executing a test suite or test cases. "*** Test Cases ***" is the section to write test cases with testing activities. The keywords and their arguments or parameters are separated by empty spaces.

In the "*** Settings ***" section, the "Resource" keyword is used to import resource files. "Test Setup" and "Test Teardown" specify preparation and finalization activities before and after executing a single test case. "Test Setup" will make "Open Browser Test Setup" to run first before running any test cases. It is a user-defined keyword called from "Common_Keywords.robot" file and it takes "http://www.sandbox.ebay.com" value as a parameter to open the tested website. For "Test Teardown" activity, "Close Window" is a keyword from Selenium2 library and will close a current open window after all testing activities are completed for a single test case. The Slenium2Library keywords are available at [12].

In the "*** Test Cases ***" section, there are three test cases "Wrong Username And Password Are Entered", "Empty Username Is Entered" and "Empty Password Is Entered". The combination of the three test cases is called a test suite. The test case names which are defined by users start at the beginning of the line and should give a general idea of what this test case is about. Inside each test case, testing activities and related information are specified. They are distinguished from test case names by being indented with empty spaces. "[Documentation]" set the related information that a user wants to document and is displayed in a result report. "[Tags]" is used to categorize test cases and users can freely tag their tests. Next, "Enter Login Page", "Enter Username", "Enter Password", "Sign In" and "Check That Login Fails" keywords are defined by users in two resource files. They are high-level keywords and contain lower-level keywords which are standard ones from the library. The goal of creating new keywords is to make the script more user-friendly, reusable, maintainable and scalable. "Enter Username" and "Enter Password" takes two parameter values, for example "user1" and "1234" in the first test case. "${empty}" is a built-in variable and carries empty value. As Listing 10 shows, all test cases have the same testing keywords and the differences are the documented text, tag and parameter values for username and

password. The keyword names need to describe their action by starting with a verb. The order of testing actions in Listing 10 is that the tool will open the login page, enter username and password, sign in and verify that a user cannot log in. Next, I will investigate the content of two resource files that contain lower-level keywords that do the actual action for login test cases.

```
*** Settings ***
Library         Selenium2Library

*** Variables ***
${Browser}      firefox

*** Keywords ***
Open Browser Test Setup     [Arguments]     ${URL}
    Open Browser            ${URL}      ${Browser}
    Maximize Browser Window
```

Listing 11. Common_Keywords.robot file.

Listing 11 demonstrates the content of "Common_Keywords.robot" file. "Library" in "Setting" section imports external "Selenium2Library". Any test files that import this resource file do not need to import "Selenium2Library" again. "*** Variables ***" specifies variable names and values whose scope stays within the current test suite only. The name and value of the variable are differentiated by empty spaces. The syntax for declaring a variable is: "${variable_name}", empty spaces and variable value. Hence, "${Browser}" variable carries "firefox" value. "*** Keywords ***" is the place where users can create their own higher level keywords based on standard ones from imported libraries. A new keyword can either have some parameters or no parameter by using "[Arguments]" after the keyword name. "Open Browser Test Setup" keyword has one parameter "${URL}" which contains the website address. The keyword includes two lower-level keywords "Open Browser" and "Maximize Browser Window". They are defined from the imported library and will actually carry out testing activities. A document of these keywords is available at [12]. "Open Browser" receives two parameter values: URL and browser type; and its function is to open the browser with a specified website address. "Open Browser Test Setup" from Listing 10 is called with a passing variable value "http://www.sandbox.ebay.com" to "${URL}" parameter in Listing

11. Therefore, "Open Browser" opens the website "http://www.sandbox.ebay.com" with the Firefox browser. "Maximize Browser Window" enlarges the open browser to full screen because "Open Browser" initially opens a smaller window. Overall, this file function is to open the tested website before running a test case.

```
*** Settings ***
Library         Selenium2Library

*** Keywords ***
Enter Login Page
    Click Link      Sign in

Enter Username          [Arguments]     ${username}
    Wait Until Page Contains Element    userid
    Input Text          userid          ${username}

Enter Password          [Arguments]     ${password}
     Wait Until Page Contains Element   pass
     Input Password     pass            ${password}

Sign In
     Click Button        sgnBt

Check That Login Fails
    Page Should Contain   Your email/username or password is incorrect.
```

Listing 12. Common_Login.robot file.

Listing 12 represents the "Common_Login.robot" script whose keywords are dedicated only for login-related testing actions. Four user-defined keywords "Enter Login Page", "Enter Username", "Enter Password", "Sign In" and "Check That Login Fails" are created. Selenium2 keywords "Click Link", "Wait Until Page Contains Element", "Input Text", "Input Password", "Click Button" and "Page Should Contain" identifies a locator on the

webpage to do the action. The most commonly used locators are text, "id", "name", "class" attributes in the HTML code.

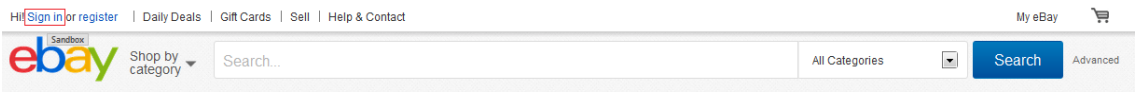


Figure 13. eBay sandbox search tool.

Figure 13 shows the search tool of sandbox eBay page which is dedicated for testing. "Click Link" clicks on the "Sign in" link text locator which is in a red square and it will lead to the login page displayed in Figure 14 below.

Figure 14. Login area with locator and used keywords.

Figure 14 demonstrates a login page with specifying "id" attribute values. The "id" attribute is unique in every page so locating an element by "id" is the best option. Continuing with explaining Listing 12, "Input Text" finds the "id" attribute that has the value "userid" which is the username field and enters the parameter value "${username}" passed from Listing 10. Similarly, "Input Password" enters "${password}" parameter value to the password filed that has "id" attribute value

as "pass". "Click Button" keyword will click on the "Sign in" button by identifying the "sgnBt" value of "id" attribute. After clicking on the "Sign in" button, the "Page Should Contain" will verify that users cannot sign in by searching for text "Your email/username or password is incorrect.". "Wait Until Page Contains Element" keyword will pause the testing process until it finds the specified locator. This keyword is necessary because I need to ensure that the keywords "Input Text" or "Input Password" can find the locator to avoid false negative test results. To find the value for locator attributes, one can right-click on target elements and select Inspect Element option for Firefox or Inspect option for Chrome. This will display the developer tool panel pointing out the HTML code of the selected element.

In summary, "Login.robot" (Listing 10) is the main script that has "Common_Keywords.robot" (Listing 11) and "Common_Login.robot" (Listing 12) as resource files. Listing 10 contains high level keywords defined by a user to describe testing steps. Each high level keyword has their lower keywords. To summarize the testing activities, Selenium2 driver opens the website window first, then runs one test case and closes the browser after finishing the current test case. It continues a similar process till the last test case. Next, I will investigate the Data Driven Development (DDD) style with item search test cases.

7.2 Item Search Test Cases

The purpose of these test cases is to check if the searched items are available in the eBay store. Figure 9 illustrates the test file structure which is similar to login tests. "Search_Item.robot" (Listing 13) is the execution file that takes "Common_Keywords.robot" (Listing 11) and "Common_Search_Item.robot" (Listing 14) as resource files.

```
*** Settings ***
Resource          ../Common/Common_Keywords.robot
Resource          ../Common/Common_Search_Item.robot
Test Template     Check That Searched Item Exists
Test Setup        Open Browser Test Setup      http://www.ebay.com
Test Teardown     Close Window

*** Test Cases ***
Search For Laptop            Acer Aspire
Search For Iphone 6          Iphone 6
Search For Watch             Watch

*** Keywords ***
Check That Searched Item Exists          [Arguments]     ${item}
   Enter Searched Item          ${item}
   Check That Searched Item Is Available
```

Listing 13. Search_Item.robot file.

Listing 13 represents the "Search_Item.robot" script content. The principal of data-driven testing is that all test cases will run one same keyword and pass different parameter values for that keyword. It means that all test cases will perform one testing activity purpose but with various data. The "Search_Item.robot" test is to determine whether the searched items (Acer Aspire, Iphone 6 and Watch) are available to buy in eBay. "Test Template" keyword in "*** Settings ***" specifies the "Check That Searched Item Exists" keyword that all test cases will call. In "*** Test Cases ***", three test cases are created "Search For Laptop", "Search For Iphone 6" and "Search For Watch" with three corresponding variable values "Acer Aspire", "Iphone 6" and "Watch". Each test case will call "Check That Searched Item Exists" keyword and pass a variable value to the "${item}" variable. In "*** Keywords ***", the keyword is defined to receive one parameter value stored in "${item}" variable and contains other two lower level keywords that are specified in "Common_Search_Item.robot" file (Listing 14). "Enter Searched Item" in Listing 13 passes the "${item}" value to the same keyword in Listing 14 and "Check That Searched Item Is Available" checks if the searched items exist. These two user-defined keywords are discussed in more detail in Listing 14.

```
*** Settings ***
Library         Selenium2Library

*** Keywords ***
Enter Searched Item          [Arguments]     ${Searched_Item}
    Input Text               gh-ac       ${Searched_Item}
    Click Button             gh-btn

Check That Searched Item Is Available
    Set Browser Implicit Wait           2
    Wait Until Element Is Visible       gf-l
    Page Should Not Contain             Refine your search
```

Listing 14. Common_Search_Item.robot file.

Listing 14 shows the script for two keywords. "Enter Searched Item" stores the parameter value in "${Searched_Item}" variable and contains "Input Text" and "Click Button" keywords. The function of these two keywords was explained in Listing 12 of the login test cases above. The keyword "Set Browser Implicit Wait" specifies two seconds as the waiting time for all keywords to repeat their action if they fail in the first time. For example, if "Wait Until Element Is Visible" cannot find the attribute value "gf-l" in the first attempt, it will wait for 2 seconds and try to search for that value for the last time before giving an error notification.

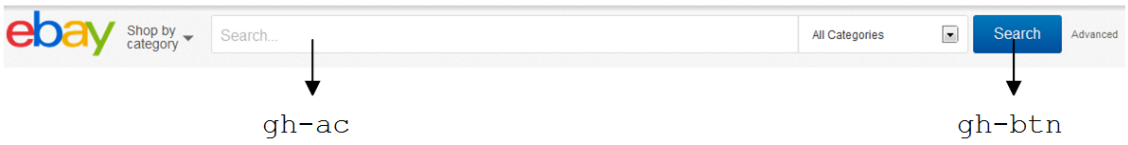


Figure 15. Search bar with id locator.

Figure 15 displays the "id" attribute of the search bar and button. "Input Text" enters text from "${Searched_Item}" value into the search bar which has the "id" attribute value as "gh-ac". Similarly, "Click Button" clicks on the Search button that has "gh-btn" attribute value. Clicking Search button leads to Figure 16.

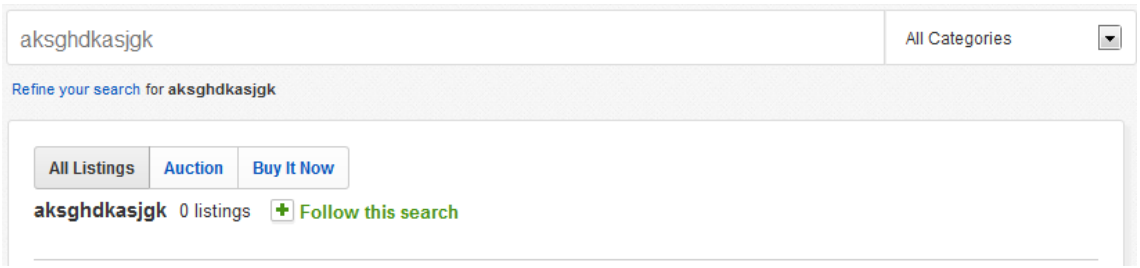


Figure 16. Search result screen.

Figure 16 represents the search result page screen when the searched item is not available. When the search results in no found items, the text "Refine your search" will appear. Hence, if the item is found, that text will not be visible. The keyword "Page Should Not Contain" checks that "Refine your search" text is not visible on the result page. Keyword "Wait Until Element Is Visible" is to make sure that "id" attribute "gf-l" appears before running "Page Should Not Contain" keyword. "gf-l" is located near the bottom of the page and if "gf-l" is visible, it means that most of the elements of the page are most likely visible for Selenium2Library. Test execution and generated reports are covered in the next section.

8 Test Case Execution Options and Reports

Robot Framework offers several methods to customize test case execution. First, to run the whole test suite, calling only the test file name is sufficient. I run Login tests with entering the command line "pybot Tests/Login.robot" into Pycharm Terminal which is opened by selecting View /Tool Windows/Terminal. Figure 15 shows the interface of the terminal.

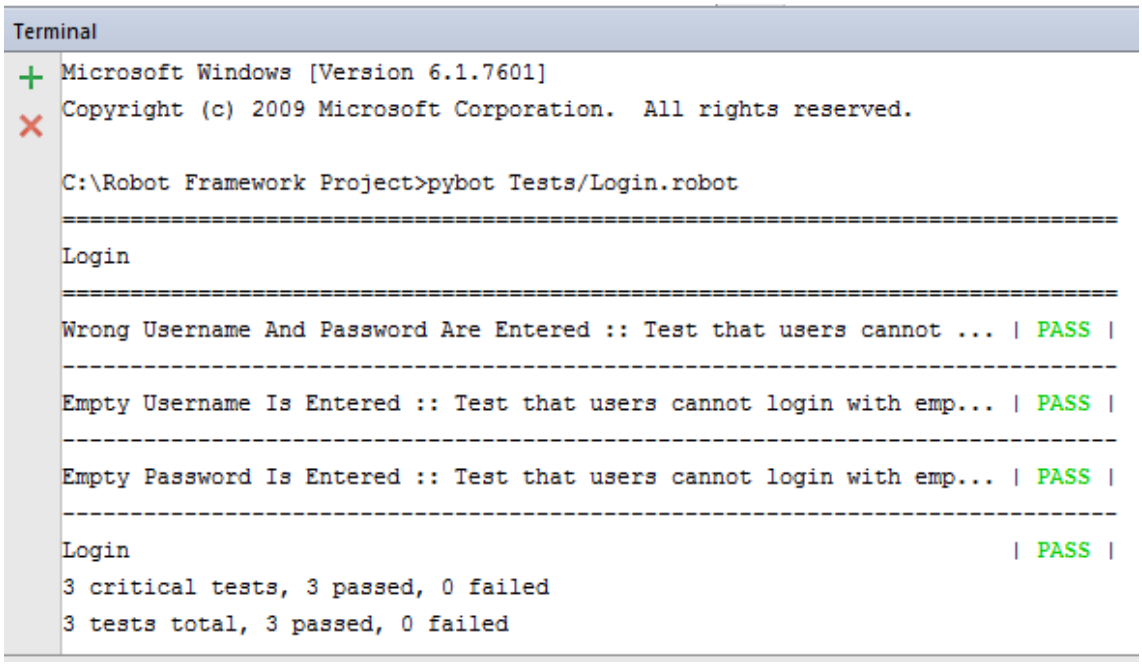


Figure 17. Running Login.robot file.

Figure 17 shows the command line to execute "`Login.robot`" file in "`Tests`" folder. The syntax of the execution command is "`pybot`" plus the path to the target file. Since the current directory is the root folder "`Robot Framework Project`", one needs to specify the full path for "`Login.robot`" as "`Tests/Login.robot`". Figure 17 also shows that all three test cases are run with the PASS result. Second, if one wants to run specific test cases, a tag can be used in the command line "`pybot --include Wrong_Credentials Tests/Login.robot`".

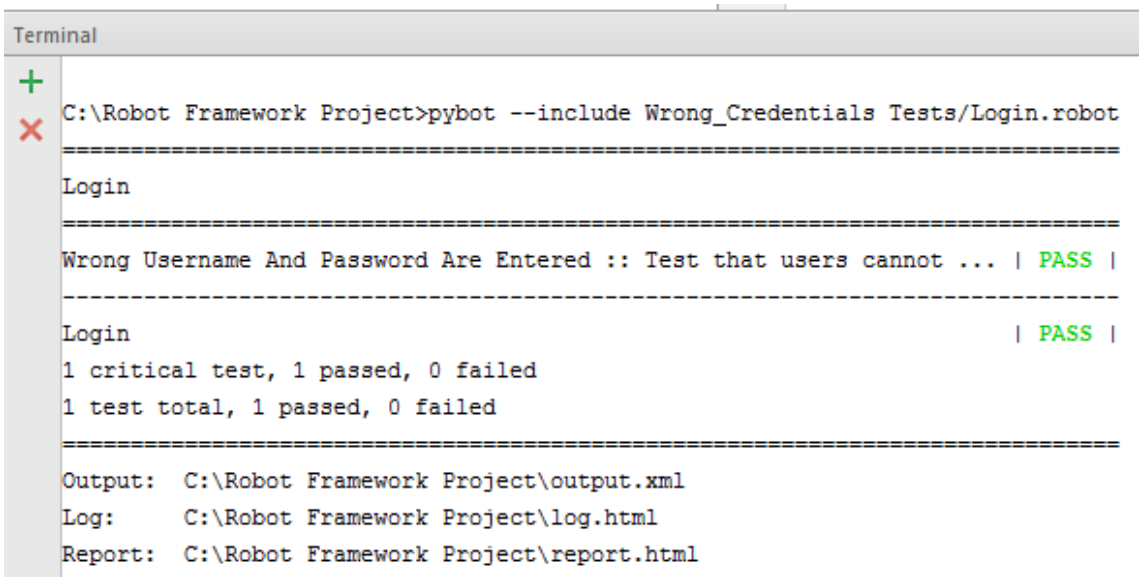
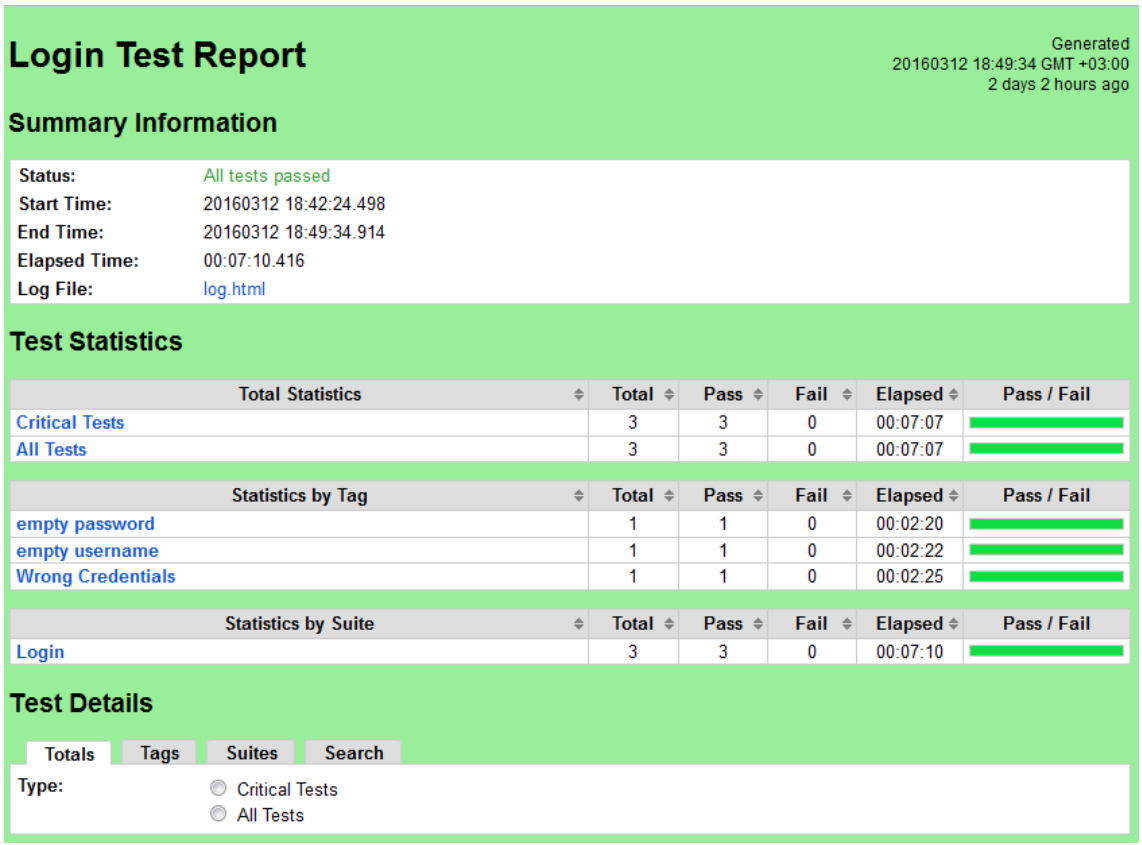


Figure 18. Running a single test case with a tag.

Figure 18 demonstrates a command line to run one test case by specifying which tag is to be executed. "`-- include`" declares the tag and only test cases that have the specified tag will be run. As Figure 18 displays, only one test case that contains the tag "`Wrong_Credentials`" is executed.

An advantage of Robot Framework is clear and detailed outcome reports. It generates report files "`output.xml`", "`log.html`" and "`report.html`" after finishing test execution. However, only "`log.html`" and "`report.html`" should be viewed for summary testing details.



Login Test Report

Generated
20160312 18:49:34 GMT +03:00
2 days 2 hours ago

Summary Information

Status:	All tests passed
Start Time:	20160312 18:42:24.498
End Time:	20160312 18:49:34.914
Elapsed Time:	00:07:10.416
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	3	3	0	00:07:07	
All Tests	3	3	0	00:07:07	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
empty password	1	1	0	00:02:20	
empty username	1	1	0	00:02:22	
Wrong Credentials	1	1	0	00:02:25	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Login	3	3	0	00:07:10	

Test Details

Totals | Tags | Suites | Search

Type: Critical Tests / All Tests

Figure 19. Summary of testing report.

Figure 19 illustrates a testing summary in "`report.html`" file. The report gives an overview of the details of the test status, start time and end time. In the "Test Statistics" section, it displays how many test cases are passed or failed. It also shows statistics sorted by a tag or suite level. A green background will be shown if all tests are executed successfully. A red background will be displayed if one test fails. In the "Summary Information" section, the general status and duration of testing are shown. If I want to

investigate more detailed data, I can click on options in “Test Details” section to see more information about each test case. In general, the layout of report.html file is user-friendly and easy to inspect data.

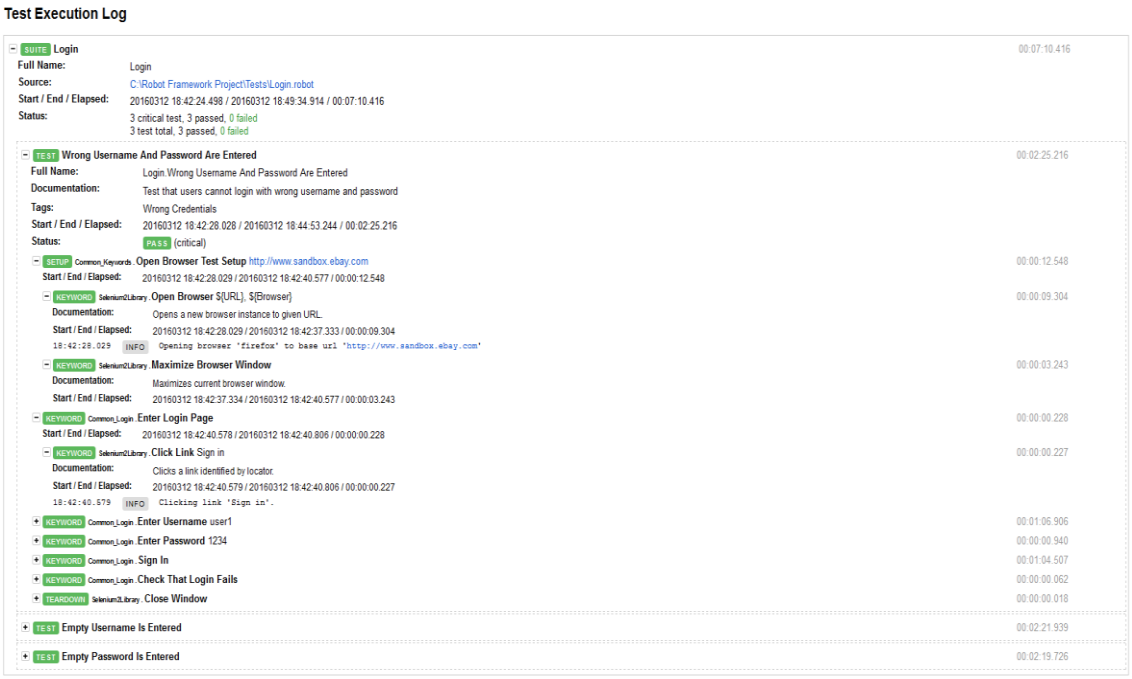


Figure 20. Detailed information in log.html file.

Figure 20 represents more details of the test results in “`log.html`” file. This report summarizes data and presents them on the keyword level. “`Wrong Username And Password Are Entered`” test case is shown with its execution keywords such as “`Open Browser Test Setup`” or “`Enter Login Page`”. It also specifies where the keywords come from. “`Selenium2Library.Open Browser`” denotes that “`Open Browser`” keyword comes from Selenium2Libray. Similarly, “`Common_Login.Enter Login Page`” implies that “`Enter Login Page`” belongs to the “`Common_Login`” resource file. In the right column, the duration that each keyword takes to perform testing activities is displayed. The green color will turn red if that keyword fails.

9 Result and Discussion

Automation testing helps save a tremendous amount of time while performing regression testing because automated tests can be run over night and at the weekend. In this project, all test cases were completed and run successfully though there were chal-

lenges and difficulties. One challenge was that some amount of time was spent for the initial effort to build reliable tests. The reliability of the tests is critical because test results can give false positive or negative outcomes.

In Listing 14, initially the keyword "Set Browser Implicit Wait" was not used and this caused false negative results with error "Element is no longer attached to the DOM". The reason was that after the eBay item search result page finished loading, it automatically refreshed the second time and at the same time the keyword "Page Should Not Contain" was running. The refreshing page made the element invalid at the time when "Page Should Not Contain" was running. For that reason, "Set Browser Implicit Wait" was needed to set an amount of waiting time that the keyword would wait before trying the last time if it failed the first time. Basically, that keyword gives time for page elements to be in a stable condition before checking for any elements. In addition, "Builtin" library of Robot Framework has keyword "Sleep" which pauses the testing execution for a specific amount of time. However, "Sleep" keyword should not be used in writing test cases because it slows down the testing speed.

Another difficulty during writing the test script was the availability of the "id" or "name" attributes. Some elements do not have those attributes so different elements are used to check for some conditions. Actually, there are different methods to locate an element such as using the XPath or CSS selector but it will complicate the test script. A lesson learned here is that when software is still in a test environment, it should be developed in a way that it facilitates automation testing such as adding necessary attributes to elements to make locating them easier.

10 Conclusion

The primary goal of the project has been achieved with utilizing Robot Framework and Selenium2Libary to write and run automated test cases successfully. The user interface of a complicated application can be tested in a much shorter period of time. In addition to GUI testing, more available libraries such as API or Database can be used to serve specific testing purposes.

Nowadays, automation testing plays an important role in the software development process because of the growing complexity of applications. It improves software quality and reduces project costs. Though automation testing brings many benefits, it cannot totally replace manual testing. Manual and automation testing need to be performed in parallel. A challenge of the project and also of automated tests is that they are vulnerable to the change of the tested software. An update to the application may cause an unexpected testing result if the structure of the tests is not well organized. So maintenance will become an issue when the number of test cases increases tremendously.

Automation testing will still keep growing in the future. Many automation tools have been created and more new and improved libraries for Robot Framework will be released to meet the needs of the automation software testing community.

References

1. Schell J. The Art of Game Design [ebook]. CRC Press; 04 August 2008.
URL:
http://proquestcombo.safaribooksonline.com.ezproxy.metropolia.fi/book/programming/game-programming/9780123694966/the-game-improves-through-iteration/ch07lev1sec4
Accessed 21 March 2016.

2. Spillner A, Linz T, Schaefer H. Software Testing Foundations, 4th Edition [ebook]. Santa Barbara, CA: Rocky Nook; 19 March 2014.
URL: http://proquestcombo.safaribooksonline.com.ezproxy.metropolia.fi/book/software-engineering-and-development/software-testing/9781492001454/3-testing-in-the-software-life-cycle/ch00lev76a_html
Accessed 21 March 2016.

3. Linz T. Testing in Scrum. Santa Barbara, CA: Rocky Nook; 2014.

4. Scrum Alliance. Learn About Scrum [online]. Westminster, Colorado.
URL: https://www.scrumalliance.org/why-scrum
Accessed 21 March 2016.

5. Evans K. Testing in Scrum Projects [online]. Tampere: Logica; 27 March 2008.
URL: http://www.cs.tut.fi/tapahtumat/testaus08/Kalevi.pdf
Accessed 03 February 2016.

6. Graham D, Veenendaal EV, Evans I, Black R. Foundations of Software Testing: ISTQB Certification. Gaynor Redvers-Mutton; 17 November 2014.

7. Ghahrai A. Fundamental Test Process [online]. Amir Ghahrai; 08 November 2008.
URL: http://www.testingexcellence.com/fundamental-test-process-software-testing/
Accessed 06 February 2016.

8. Bath G, McKay J. The Software Test Engineer's Handbook. Santa Barbara, CA: Rocky Nook Inc; 2008.

9. Burnstein I. Practical Software Testing. New York, USA: Springer; 2003.

10. Nokia Solutions and Networks. Robot Framework User Guide Version 3.0 [online]. 31 December 2015.
URL: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html
Accessed 20 February 2016.

11. Python Software Foundation. robotframework 3.0 [online].
URL: https://pypi.python.org/pypi/robotframework
Accessed 24 February 2016.

12. Robot Framework. Selenium2Library [online]. 26 August 2015.
URL: http://robotframework.org/Selenium2Library/doc/Selenium2Library.html
Accessed 06 March 2016.