Designing a Software Test Automation Framework

Sabina AMARICAI[1], Radu CONSTANTINESCU[2]
[1]Qualitance, Bucharest
[2]Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies, Romania
sabina.amaricai@qualitance.ro, radu.constantinescu@ie.ase.ro

Testing is an art and science that should ultimately lead to lower cost businesses through increasing control and reducing risk. Testing specialists should thoroughly understand the system or application from both the technical and the business perspective, and then design, build and implement the minimum-cost, maximum-coverage validation framework. Test Automation is an important ingredient for testing large scale applications. In this paper we discuss several test automation frameworks, their advantages and disadvantages. We also propose a custom automation framework model that is suited for applications with very complex business requirements and numerous interfaces.

Keywords**:** *Software Testing, Test Automation, Test Automation Framework, Data Driven, Keyword Driven, Hybrid*

1 Introduction

Software testing has been a rapidly growing industry in the past ten years. Nevertheless, the domain is new so there is a lot of room for improvement and a lot of room for innovative ideas. A very interesting and challenging chapter of software testing is software testing automation. This falls somewhere between software testing and software development, using both programming concepts as well as testing ones. Diving further into software automation, one of the biggest challenges is to keep the testing perspective while codding, as the independence of testing versus development is an extremely important principle.

We see a lot of improvements in Software test automation in the past five years. As it happens in any growing industry, there were set a lot of trends. It usually starts with record and play approach and evolved to a modular approach and moving towards the data-driven and keyword driven. Of course, these trends started a lot of debates on which design is better or more suitable for your team, your business and your needs.

Before we start discussing test automation design, we will define some of the most common terms related to this topic. Software testing is defined as "an investigation conducted to provide stakeholders with information about the quality of the product or service under test" [3]. Therefore, the main goals of this activity are to detect and prevent defects as well as to insure the intended behavior of the tested software [5], [6], [7].

Software test automation represents the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions (BS 7925-1) [1].

Test Automation Framework represents a framework used for test automation. It provides some core functionality (e.g. logging and reporting) and allows its testing capabilities to be extended by adding new test libraries [1]. "An automated test framework may be loosely defined as a set of abstract concepts, processes, procedures and environment in which automated tests will be designed, created and implemented. In addition, it includes the physical structures used for test creation and implementation, as well as the logical interactions among those components" [2].

A test automation framework has the structure of a software application. As an application, a test automation framework defines common functions such as handling external files, GUI interaction, provides templates for test structure, and therefore developing an

DOI: 10.12948/issn14531305/18.1.2014.14

automated solution is very similar to developing software applications [3], [7], [9], [10], [11], [12], [13].

In any type of design of software test automation framework, in order to test an application, we need the following:

- *Test case or test flow* – The definition of a test case in automation is the same as the test case defined by ISTQB: "A set of input values, execution preconditions, expected results and execution post-conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement" [After IEEE 610]. In other words, the test case represents the sequence of steps followed to test a specific functionality.
- *Test script* - A test script is a set of instructions (written using a scripting/ programming language) that is performed on a system under test to verify that the system performs as expected. Test scripts are used in automated testing.
- *Test data* – the input used to test – the data used to test specific functionalities of the application, such as user data, search queries, expected messages in case of invalid input, etc.
- *Locators* – identifiers for the application elements such as buttons, input fields, alerts, etc.

The difference between the various types of test automation framework designs is usually based on how and where the test case, test data and locators are defined. Depending on the layer where the three elements above live, the automation framework implementation requires a different level of abstraction (generalization), therefore different programming skills for the team that develops and maintains the framework [6], [7], [8], [9], [10]. For example, *HP QuickTestPro* uses an object repository where the elements are stored (the locators), *IBM Rational Functional Tester* offers a similar object repository or a custom API to define custom object repository and *Selenium*, in case of selecting record and play functionality, uses the locators in the test scripts [11], [12], [13].

There are a couple of test automation framework types but the most common are: data – driven, keyword – driven and hybrid. In a data – driven automation framework, the test data is stored in external files or database. Its biggest limitation is the fact that a test script can only execute similar tests, therefore, new scripts need to be developed when new test cases have to be created. This type of framework is commonly used with applications that require testing with a large amount of data on similar scenarios [11], [12].

A keyword – driven framework extends the idea used by data driven frameworks. Now, not only the test data but also the actions on the application elements/objects are stored in external files. This approach makes it easier for the test engineers to create test cases without ever touching the framework code. The test data is still read from external files as in data-driven testing. As Fewster and Graham (1999) put it, keyword-driven testing is a logical extension to data-driven testing [1], [2], [3], [4].

Implementing a keyword driven framework requires a lot of programming skills and a high level of abstractization. However, creating new test cases is done easily, by teams without any programming skills. This type of framework fits a broader range of applications but it is usually limited only by the technology used to implement it. Of course the idea can be implemented using various technologies, depending on intended use [7], [8]. [13].

In a hybrid framework, the basic concepts of data driven and keyword driven are combined. This type of automation framework can accommodate easier various types of applications and clients requests. It requires less generalization, compared to the keyword driven framework but it still allows more flexibility than the purely data driven one [1], [2], [3], [10].

2 Existing Test Automation Designs

Given so many automation frameworks designs, one would have to choose one design

DOI: 10.12948/issn14531305/18.1.2014.14

that better suits its needs. Before choosing any idea of automation framework, it is very important to define the requirements the project has. Let's take for example some of the most often requirements we have encountered while implementing an automation framework:

- Flexible and reusable automation framework for multiple applications.
- Company's test team has little to no programming skills.

In this article we are going to focus on two designs of software testing automation frameworks that answers to the requirements described above: *PageObject* design and a custom framework developed as a new automation solution for several projects.

These days, the one of most discussed trend in software testing automation is the *PageObject* design, which is very well supported by *Selenium 2.0* offering the *PageObject* pattern. In *PageObject* design, the objects define each application page or section that is displayed on more than one page (such as header, footer, etc.). Then, on each page element (such as button, input text, etc.) there are actions defined. The definition of the actions can be done with the same method as for the element definition. The test scripts (that represent the actual test cases) use one or more objects as shown in Figure 1.

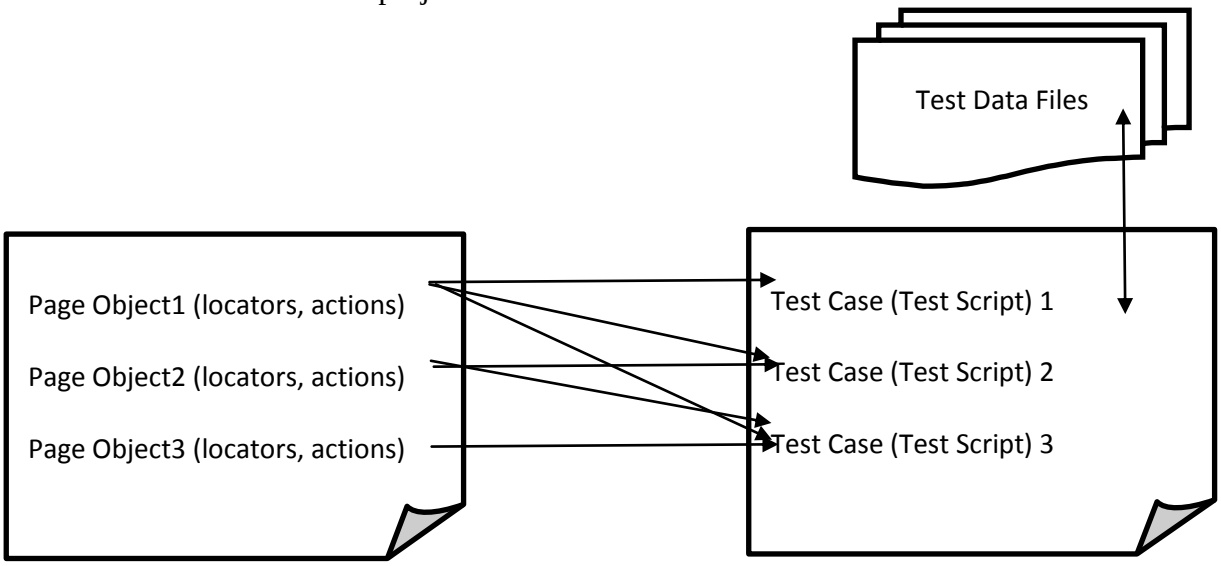

Fig. 1. *PageObject* design

Let's consider the login screen of an application to describe the two test automation designs proposed. As shown in Figure 2, the login screen (or page) is displayed after the user clicks on the *Navigate to Login* button. The login screen contains the following elements:

- *Username field* – it is a field where the user enters text (the username to access the application) – it is identified using css selectors by the field id - username (the css selector will be: #username)
- *Password field* – it is a field where the user enters text (the password to access the application) – it is identified using css selectors by the field id - password (the css selector will be: #password)
- *Login button* – it is a button that submits the login form – it is identified using css selectors by the field id - submit (the css selector will be: #submit)

DOI: 10.12948/issn14531305/18.1.2014.14

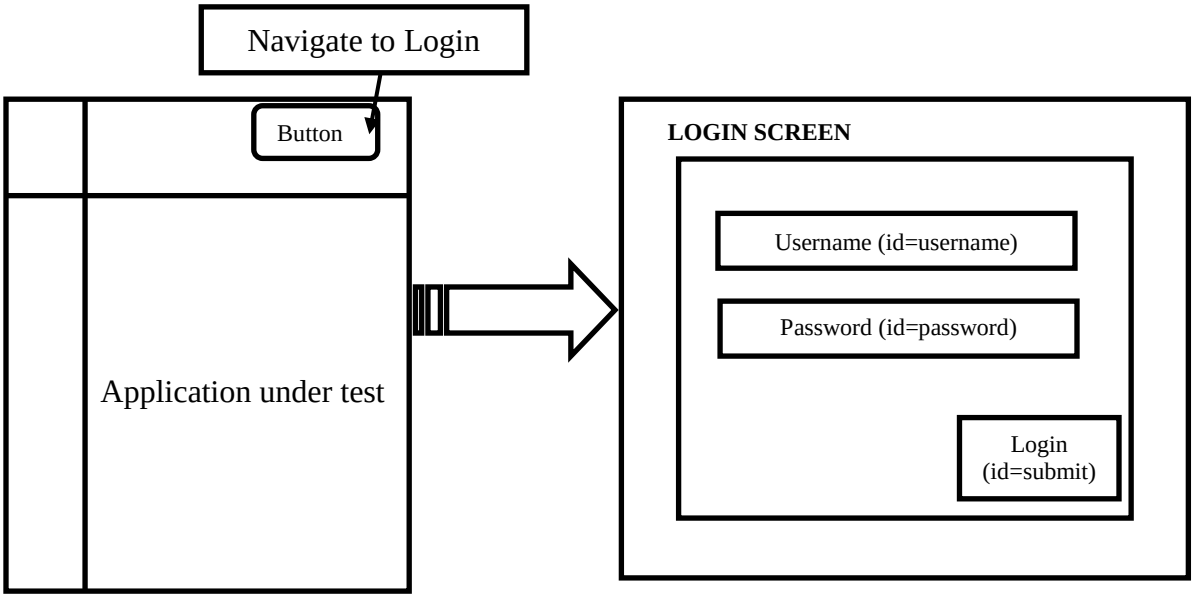

Fig. 2. Application under test – login screen

For example, for the login test, this approach would go like this:

1. Define login screen: *LoginPage*
    a. Username field (input text with a specific locator)
    b. Password field (input text with a specific locator)
    c. Login button (button with a specific locator)
2. Define actions for the elements above:
    a. Enter text in username field (`enterUsername (parameter: username)`)
    b. Enter text in password field (`enterPassword(parameter: password)`)
    c. Click on login button (`clickLogin(parameter: username)`)
3. Define test script:
    a. Navigation to login screen – generic description of a method that performs the navigation to the login screen – in this example – click on *Navigate to Login* button (defined in the login page object or in a different object)
    b. `LoginPage.enterUsername("user")` - call to the action to enter text in username with a parameter for the username text
    c. `LoginPage .enterPassword("pass")` - call to the action to enter text in password with a parameter for the password text
    d. `LoginPage .clickLogin()` - call to the action to click on the login button
    e. Validate the message or the new screen - generic description of a method that performs the validation of the message displayed after login is successful or a validation that a specific element is displayed on the next page – such as *Logout* or *MyAccount*.

In this design, if the page or screen changes, the updates are required only in the page object of the specific screen or page – on the object definition part. In order to write the test scripts this way, the team responsible with test case maintenance and test suite augmentation is required to have a low to medium level of programming skills. Let's consider the parameters read from an external file or a database. In this case, one test script as defined above will be able to execute only similar test cases.

We consider this approach similar with the one *HP QuickTestPro* implements with their object repository.

DOI: 10.12948/issn14531305/18.1.2014.14

Since *PageObject* approach was introduced in *Selenium* it became more and more popular. As any design, the *PageObject* one can be implemented using almost any automation tool on the market. We see a lot of the test automation migrating towards this design disregarding completely the applications tested or the test team involved in the test automation maintenance process. Depending on the purpose served by a specific test automation framework the *PageObject* design can ease the work.

The *PageObject* design, compared to the most common automation approach – starting from the tool recorded tests, reduces the code redundancy by implementing the elements and actions in the page objects. When someone has to enter text in the username field there is simply a call to the method that performs this task instead of identifying the element and defining the action each time it is needed. Locators and page actions are stored in a unique location, when something changes in this area there is a single place where updates are required. Also, tests that call the *PageObject* methods and that makes them easier to read; they can be used as documentation as well.

3 Proposed Automation Framework Design

We consider *PageObject* automation model a good solution if the test team who maintains the test automation framework has medium programming skills and there are one or two different applications tested with this approach – having in mind the number of page objects that need to be created for each page or section. Also, it would be easier and much more maintainable if the number of distinct pages (or sections that appear on more than one page) is relatively small (10-20).

In this context we may raise the following questions:

- What if the quality assurance team has little to no programming skills as we have all seen?
- What if the company is a bank that has more than 10 different applications (web and otherwise) that require test automation?
- What if the company that implements e-commerce web applications for more than 10 different clients?

In these cases, we wouldn't consider *PageObject* design the best way to go because the number of page objects will increase and the maintenance would become difficult, the number of tests will increase beyond maintainability and most painful, there would be no experience in the team to perform the maintenance and to develop new tests. Now, depending on the test team that would have to maintain the test automation framework, we would consider the following options:

1. Keep the logic and the test flows in the test framework code and use external files for the locators and test data.
2. Or go even bolder and use a higher level of generalization in the code and keep the locators, the test data and the tests logic and flows in external files.

Both this approaches require a high level of programming skills for the team that develops the framework but little to no programming skills on the test team that maintains the test automation framework. We are not going to discuss at large any of the ideas above, that being the topic of a different article, but we would describe a bit the concept behind those types of test automation framework implementation as shown in Figure 3.

DOI: 10.12948/issn14531305/18.1.2014.14

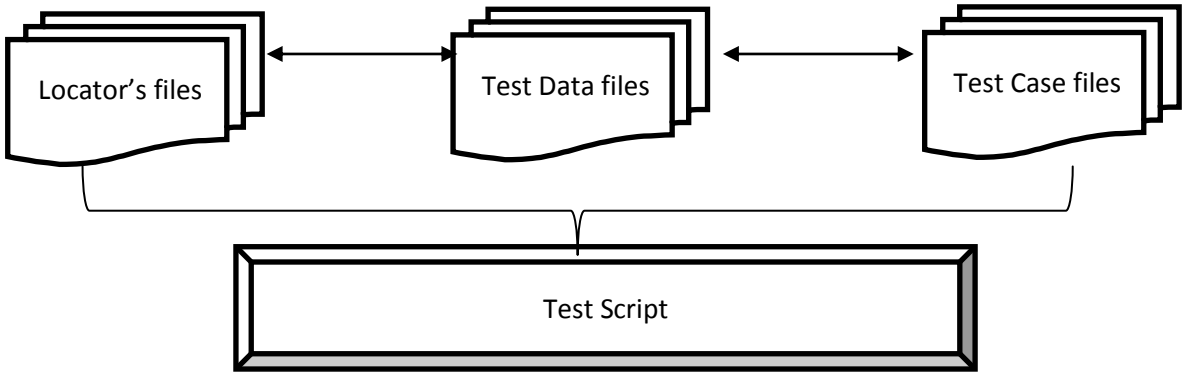

Fig. 3. Proposed automation framework design

In the second case, we would consider three types of files:

a. *Locators*: file that contains the locators of the application (split by pages, sections, to make it easier to read and search)
b. *Data*: file that contains data sections: we include here the input, asserts. Each section has the order specific to the application – split by reusable application sections
c. *Test*: file that contains tests, each test containing calls to the data sections and make the test flow

To make the maintenance easier, we have implemented this approach using the sequence in the test and data file and asking the locator's files for the locators of the elements required. In this manner, if a locator gets deprecated or the order is not right there are no actions required. Also, if a locator is missing and it is required by a test, there is an error logged.

Basically, the automation design proposed contains three types of files: locators, test data and test case in a tabular format with different headers. Locators file is following the convention: <*section/page name, element name, element locator, locator type, action on the element*>. In this file, the order of the locators has no relevance what-so-ever. The lines here can be out of order but the recommendation is to keep one to make it easier to read and use.

- *Section/page name* – it is a user chosen name used to identify a part of the application tested. This name will be used in mapping the types of files as it is described in details in the next paragraph. This name should be unique.
- *Element name* – it is a user chosen name used to identify an element of the page. This is also used in the mapping of the files but also for the users to understand what of the page elements is defined there. This name has to be unique per section name.
- *Element locator* – is the application element locator. This locator can be defined based on the application DOM, if we are talking about web applications and can be identified using any of the available methods: xpath selectors, css selectors, id, name, depending on the technology used to implement this framework design.
- *Locator type* – in order to make the framework as flexible as possible, depending on the technologies used to implement it, it is useful (and sometimes mandatory) to specify the type of selector used to identify the element – such as css selector, xpath, etc
- *Action on the element* – defines the action type that can be performed on the specific element. The actions have to be defined in the framework implementation and used as they are defined. For example, *inputText*, the action that we are going to use in the example below is implemented to input text on an element. This should be used for inputs or edit boxes.

Test data file is following the convention: <*data section name, element name, test data value*>. In this file, the order of the data in

DOI: 10.12948/issn14531305/18.1.2014.14

the same data section is used when executing the test cases in the sequence of steps followed while testing a specific functionality. Otherwise, the sections themselves can be in any order the user chooses.

- *Data section name* – it is the user chosen name for a data section. The data section name should follow the naming convention: section/page name (from the locators file) plus "_", plus name to describe the data (such as valid scenario, invalid, etc.) as we will present in the examples below. The section/page name is mandatory and the "_" and what follows is optional. This naming convention is used in the mapping of the data with the locators. This name also has to be unique.
- *Element name* – it is the same user chosen name described in the locators. This is also used for mapping the files.
- *Test data value* – it is the actual test data as described in the beginning of the article. It can be a text entered in an input field or a message that is expected on the page after an action. To keep the file columns number the same, for the elements that don't require data, any character or data can be used but it will be disregarded when performing the action: for click on buttons – there is no data required but "click" should be added in this file to keep it consistent and easy to read.

Test case file is following the convention: *<test name, data section name>*. In this file, the order of the data section names in the same test name section is used when executing the test cases in the sequence of steps followed while testing a specific functionality. It is recommended to have also an order in the test name sections because the test cases will be executed in the order written in the file.

- *Test name* – it is a user chosen name for the test case. It has no link to the other files but it has to be unique.
- *Data section name* – it is the same data section name described in the test data structure.

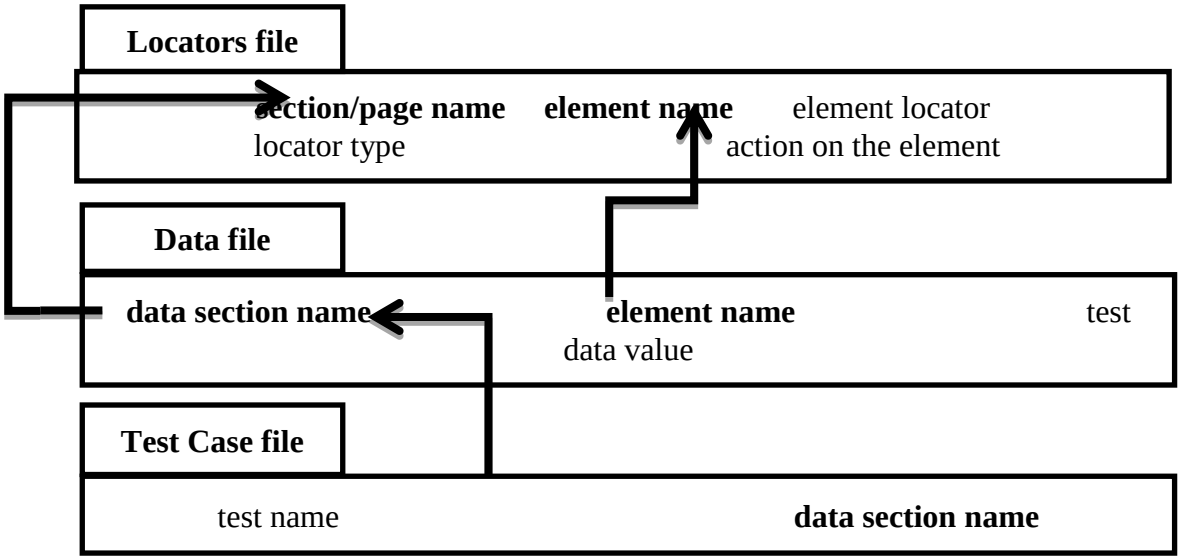

Fig. 4. Files mapping model for the proposed automation framework

The files are linked together as shown in Fig.4, using the convention described above. The test case uses the data section name to link to the test data file. This means that when a test is executed, each data section mentioned in the test case will be looked up in the test data files and the entire sequence of steps will be executed in the order written in the test data file for that section.

The test data files are using the section/page name and the element name as a key to link to the locators file. For each step of the test data section, the elements locators and actions that have to be performed on each of

DOI: 10.12948/issn14531305/18.1.2014.14

the elements will be looked up in the locators file. This time, the mapping between the test data and the locators' files uses two keys to allow a better flexibility. For example, in the login and registration forms, there will be username and password present but on different pages and most certain with different locators. Using the two keys to map these files allows the user not to worry about the uniqueness of the element name.

As an example, for the login test, this approach would go like this:

1. Locators: (for the login section/screen : login)
    a. *<login, username, #username, css, inputText >*– section name, a name or identifier of the locator , locator for username, and the type of action performed on this element like inputText
    b. *<login, password, #password, css, inputText >*– section name, a name or identifier of the locator , locator for password, and the type of action performed on this element like inputText
    c. *<login, login, #submit, css, click>*- section name, a name or identifier of the locator , locator for login button, and the type of action performed on this element like click
    d. *<login, login, #message, css, validate>* section name, locator for the message that login was successful or unsuccessful, a name or identifier of the locator and the type of action performed on this element like validation
2. Data: (data section for a positive login scenario: *login_validLogin*)
    a. *<login_validLogin, username, user>*– Section name, the identifier of the element to map it with its locator and valid username
    b. *<login_validLogin, password, pass>* – Section name, the identifier of the element to map it with its locator and valid password
    c. *<login_validLogin, login, click>* - Mention of the login button so the action described in the locators would be performed on the login button
    d. *<login_validLogin, message, Login was successful>* - The text that should be displayed in case of a successful login
3. Test:
    a. *Navigate_loginPage* - Navigation to the login screen (is a section in the data file that should contain the action to click on the Navigate to Login button)
    b. *login_validLogin* - Call to the login section described above

Test files would contain the negative tests for login as well, by calling a different data sections. We will present a short description for the negative tests, those test can be further adapted to the pattern conventions we have previously described. The data sections for negative scenarios for login should be more modular and split as follows:

a. *invalidLoginData1*: invalid username, invalid password
b. *invalidLoginData2*: valid username, invalid password
c. *validateInvalidLoginMessage*: validate invalid login message

In this case, the negative login tests would look like:

- *TestInvalidLogin1*:
    a. TestInvalidLogin1, Navigate_loginPage
    b. TestInvalidLogin1, invalidLoginData1
    c. TestInvalidLogin1, validateInvalidLoginMessage
- *TestInvalidLogin2*:
    a. TestInvalidLogin2, Navigate_loginPage
    b. TestInvalidLogin2, invalidLoginData2
    c. *TestInvalidLogin2, validateInvalidLoginMessage*

Using this type of approach, the test files and most of the data sections are the same for any application that uses a login with username,

DOI: 10.12948/issn14531305/18.1.2014.14

password and clicks on a login/submit button. And, going a bit further than the common login example used throughout the article, considering the registration form that requires more information filled in, the files won't be very different; there will be just a couple more lines, keeping the described structure. The test team that creates and maintains the tests will only interact with the text files following a minimum set of rules, described in the files mapping model. Of course, this is one of the implementation ideas for this design. The set of files can be mapped in various other ways, depending on the types of files used.

4 Conclusions

Some of the most common reasons to use automation in the testing process is to execute a set of tests much faster and avoid repeating manual testing. Also, it makes it easier to deploy frequent builds and helps increase the confidence in the developed application. While executing the set of automated scripts, the test team could focus on different areas of the application.

In order to achieve all that, one has to choose a test automation framework to meet the company's needs. As we mentioned in the beginning of the article, most of the companies now require a framework flexible and reusable across multiple applications that requires little to no programming skills from the internal test team. To meet this needs and make the best of it, we came up with one solution that is already implemented in a couple of companies – *PageObject* design that proved to have minor disadvantages when it came to create new test case and maintain the existing ones due to the little to no programming skills in the test team.

We also proposed a new solution that comes really close to a keyword driven approach that uses external files to store the test cases, test data and the application locators. All that the test team has to do when creating a new test case, is defining one or multiple data sections and create a new test case section in the test case file. If the elements are not defined yet in the locators' files, they have to create a section or just the elements there as well.

We have also identified several improvements points for the custom solution. As described above, there is one action that can be performed on an element of the page, as the solution is now. What if there are multiple actions that need to be performed on an element? For example, on a button, one can perform actions like click the button, validate the text displayed on the button, etc. In the solution described above, there is a workaround for this example but it involves redundancy in the locator's files. The element has to be defined twice on the same section with different element names and different action, but with the same element locator. To improve this solution, instead of defining one action per element, there can be defined a set of actions. Of course, there might be the problem of the order of the actions performed: first you validate the text or click the button? We could avoid this question by setting the order of the actions in the file like it should be performed in the application. We intend to follow and extend those ideas in a further article.

References

[1] P. Laukkanen, "Data-Driven and Keyword-Driven Test Automation Frameworks"

[2] http://www.automatedtestinginstitute.com/

[3] A. M. Memon, M. E. Pollack, and M. L. Soffa, Using a goal-driven approach to generate test cases for GUIs. In ICSE '99: Proceedings of the 21st international conference on Software engineering, pages 257–266. IEEE Computer Society Press, 1999.

[4] M. Fewster and D. Graham, *Software Test Automation*. Addison-Wesley, 1999.

[5] I. M. Iacob, R. Constantinescu, *Testing: First step towards software quality* - http://jaqm.ro/issues/volume-3,issue-3/pdfs/iacob_constantinescu.pdf

DOI: 10.12948/issn14531305/18.1.2014.14

[6] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, 2002.
[7] I. Ivan, C. Boja, *Practica optimizarii aplicatiilor informatice*, Editura ASE, Bucuresti 2007, 483 pg, ISBN 978-973-594-932-7
[8] I. Ivan, P. Pocatilu, *Testarea software orientat obiect*, Editura INFOREC, Bucuresti, 1999, 194pg, ISBN 973-98508-0-4
[9] C. Kaner, "The Ongoing Revolution in Software Testing," *Software Test & Performance Conference*, Baltimore, MD, December 7-9, 2004
[10] M. Fewster, *Software Test Automation: Effective Use of Test Execution Tools* (Paperback)
[11] D. Graham, M. Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*
[12] L. Kanglin, *Effective Software Test Automation: Developing an Automated Software Testing Tool* (Paperback)

Sabina AMARICAI is software testing consultant at Qualitance, she has received her Master Degree in Information Security from the Faculty of Cybernetics, Statistics and Economic Informatics in 2011 and her Bachelor Degree in Computer Science from University of Bucharest, Romania in 2008. She has more than three years of developer's experience and more than five years' experience in software testing focusing on automation and performance testing. Her fields of interest include software testing: automation and performance and information security.

Radu CONSTANTINESCU has a background in computer science. He has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2003. He defended his Ph.D. in 1998, with a paper on Information Security. He has published as author and co-author over 60 articles in journals and national and international conferences and over 10 books. His fields of interest include information security, software testing and operating systems designs.

DOI: 10.12948/issn14531305/18.1.2014.14