Business Simulation Test Automation applied to Portuguese Social Security

Hugo André Martins Rocha

1 Instituto Superior Técnico, Universidade de Lisboa
hugo.rocha@seg-social.pt

Abstract - Since many companies are doing test automation, you’d think it would be a simple task, but there are challenges with incorporating test automation into your development cycle. Two of the biggest challenges are the time and the skills needed to implement and maintain the test scripts. Some organizations have started to look ahead for the ROI (Return of Investment) and stopped doing automated tests, considering these difficult challenges.

This paper presents an innovative solution to address these challenges as it works like a RAD (Rapid Application Development) model applied to the automated test scripts. As this solution defines and generates a flexible class model easy to implement and change, it reduces the effort of building and maintaining the test scripts. It uses BPMN (Business Process Model and Notation) that document the functional specification of an application and from that graphical diagram automatically generates all possible automated test scripts. Furthermore, it is also tool independent if Java language is used.

Keywords: Test Automation, test scripts, Keyword-driven testing, BPMN, Test Cases, Use Cases

I. INTRODUCTION

Over the past few years, several organizations with big and complex development cycles started to implement test automation as part of their continuous integration process. These automation tests are of particular importance and relevance to ensure that every daily build doesn’t “break” and continues to work for the previous functionalities as it did, prior to the changes. This is called regression testing [2].

Big software houses such as the Instituto de Informática, I.P. have complex and different software architectures in order to implement solutions for its clients. One of its main clients is the Portuguese Social Security, also known as Instituto da Segurança Social. Every year, more than 1300 new versions are developed and tested from an overall of more than 100 applications. From these 1300 new builds, 70% are related to maintenance versions and due to regression testing, which plays a very important role here. Regression testing is massive and for that reason, investing in automating regression tests is very important in this scenario. In addition, there are many data migration projects which need data validation for millions of records. Automated tests, with pattern matching, are fundamental for these migrations tests, and due to regression testing, which plays a very important role here. Regression testing is massive and for that reason, investing in automating regression tests is very important in this scenario. In addition, there are many data migration projects which need data validation for millions of records. Automated tests, with pattern matching, are fundamental for these migrations tests.

In the last paragraph it is possible to understand that automation testing can play a very important role in the following areas of the Portuguese Social Security: a) Regression Testing; b) Volume Testing; c) Data Migration Testing. The process of implementing these automatic test scripts is done by means of an automation testing tool that records the user actions, mouse clicks and keyboard inputs, and generates in the end a test script, which can be played to execute the automatic test. It is also possible to create data pools in order to be possible to use different data during the execution of these automation tests. In Social Security the automation test tools used are: Selenium¹, an open source tool, and SilkTest², a commercial tool from Micro Focus®. Both use Java language as the test script language. These tools and most of other automation test tools, individually, miss important features, like: (1) A global and centralized repository for storing all the test scripts; (2) Control of test scripts versions; (3) Parameters in the scripts to use different test environments variables; (4) Detailed and centralized reports of the tests ran; (5) Schedule runs; (6) Possibility to run the tests in remote machines using a test farm of agents; (7) Utility libraries to support functionalities such as SQL queries on the database, notifications by mail, screenshots and video recording. To overcome these missing features on automation testing tools, in general, a framework of automation tests was implemented two years ago, using Selenium [10] and SilkTest as the engines to record and execute the scripts. This testing framework is called TestAut.

Nevertheless, even with this framework, implementing and maintaining automated test scripts can be very time-consuming and skills-demanding. The development process used in Instituto de Informática, I.P. is UP (Unified Process) and the analysis documentation is based on Use Cases and Business Rules
defining the Test Cases and recording the test scripts is time consuming and demands lots of concentration in order not to miss a particular test path or combination, because test documents are being used.

This approach contributes by giving a step further into automation testing, expanding and complementing the Portuguese Social Security automation testing framework (TestAut). It is based on an approach followed by a former student in Link Consulting®, by means of keyword-driven testing from BPM (Business Process Management) diagrams [12], which provides the capability of specifying the business processes with a graphical notation [7]. This will facilitate the perception of the processes since it is easier to understand a graphic diagram than text specifications and will allow the implementation of a parser that automatically generates the test scripts and, consequently, the Test Cases (since a test script is the implementation of a Test Case).

In this paper, tests were conducted to find the best solution to

1  http://www.seleniumhq.org/
2  http://www.borland.com/en-GB/Products/Software-Testing/Automated-Testing/Silk-Test
simplify and to accelerate the process of test automation. As a result, the 3 main objectives established were:

1. Define a conceptual approach to model the business through the use of the BPM specification; (2) Build an architecture that will integrate with the test automation framework (TestAut) that already exists, adding the necessary new components to expand and improve this solution; (3) Do a proof of concept to test and evaluate this approach, by choosing a process (a real Use Case and set of Business Rules) and implement the automated tests for this process. In the proof of concept, Use Cases and Business Rules are transformed into BPM (Business Process Management) diagrams and test scripts are automatically generated for each path of that diagram. The test scripts are written using reusable methods (keywords) corresponding to the activities (tasks) in the diagram.

The main contributions of this work are: (1) a new approach to design Use Cases and Business Rules using the BPMN diagrams; (2) automatically generate tests scripts with all possible scenarios, from the BPM diagrams; (3) Automation Testing Tool independent if Java language is used.

II. How Automation Testing Tools Work

Manual testing is performed by a person sitting in front of a computer carefully executing the test steps. Automation Testing means using an automation tool to execute your Test Case suite [1]. The automation software can also enter data into the system under test, compare expected and actual results and generate test reports. By using a test automation tool it’s possible to record this test suite and replay it as required. Successive development cycles will require execution of same test suites repeatedly. Once the test suite is automated, no human intervention is required. Nevertheless, Test Automation demands considerable investments of money and resources [3, 5].

There are a lot of reasons as to why automated testing is beneficial and by using these best practices in your automated testing, you can ensure that your testing is successful and you get the maximum ROI. Thorough testing is crucial to the success of a software product. If your software doesn’t work properly, chances are that most people won’t use it…or at least not for long. But testing to find defects – or bugs – is time-consuming, expensive, often repetitive and subject to human error. Automated testing, in which QA (Quality Assurance) teams use software tools to run detailed, repetitive, and data-intensive tests automatically, helps teams improve software quality and make the most of their always-limited testing resources. Test Automation tools help teams test faster, allows them to test substantially more code and improves test accuracy [4, 6]. This is particularly important and translates into much more benefits when used for automating regression testing, which would require repetitive manual tests to be done over and over again, each time a new build of a particular application is delivered for testing. If these regression tests were done manually an enormous amount of time would be spent in these activities. Automating these repetitive tests will free up QA engineers so they can focus on tests that require manual attention and their unique human skills.

First, and before you use a tool, you need to identify and define the Test Cases that you want to automate. Second, you need to choose the right tool for your development environment and architecture. Leaving important aspects aside, like price and support, and since these tools need to be compatible with the GUI (Graphical User Interface), the development language and technologies used to render the GUI have an impact significance when choosing the right tool. For instance, if your GUI is in Java® or .NET® you need to look for automate testing tools that work well with these technologies. If you are using SAP® maybe you will need a different tool and so on. After choosing two or three tools that are compatible with your GUI environment, a proof of concept to choose the final one should be considered for your final decision.

III. The Approach

This section presents this approach, starting with an overview of how it works (Section III-A), then detailing its several aspects in the rest of the sections.

A. Overview of the approach

The architecture proposed addresses the several activities involved in the test discipline, from test planning to test execution. This solution was built from scratch, in terms of methodology and processes implementation. Its main goal is to improve the efficiency and the quality of the testers work. In order to do that, a new methodology based on the BPM modeling is proposed which involves the following steps:

1- Representation of the Use Cases and the Business Rules in BPM. At the moment, the test cases are designed manually by following each of the flows in the Use Cases and identifying, for each flow, the test conditions and the test results (pre-conditions and post-conditions). The Test Cases are then the combination of all of these test conditions. In this way, the Test Cases design can take a long time and it’s easy to miss an important combination.

In this new methodology, the Use Case is represented in a BPM diagram where each flow is modeled using activities and gateways. The activities represent all the actions and the gateways all the decisions. Each path in the diagram will represent a Test Case which results in a much simpler way of extracting the Test Cases, automatically, without human interaction.

2- Running a XPDL (XML Process Definition Language) Parser that automatically generates the test scripts for all the paths and the Test Library with all the keywords.

Having the BPM diagram designed, it is thus possible to export the diagram to a XPDL file [19]. This is a XML document that is part of the BPM specification which represents the BPM diagram where each element (activity, gateway or transition) is identified by a unique id. Since each id is unique (for example, e5ade915-6533-4fe3-b1bf-6702782dac1f), it can be used as a keyword in the test scripts to identify the activities. This way, the keywords can then be transformed to Java methods to be executed by the test scripts. Because Java has defined rules to the names of the methods, this id is modified in the following way by the parser: - It appends the “activity.” string at the beginning (Java methods can’t start with numbers and since the id is a hexadecimal string

2- Running a XPDL (XML Process Definition Language) Parser that automatically generates the test scripts for all the paths and the Test Library with all the keywords.

Having the BPM diagram designed, it is thus possible to export the diagram to a XPDL file [19]. This is a XML document that is part of the BPM specification which represents the BPM diagram where each element (activity, gateway or transition) is identified by a unique id. Since each id is unique (for example, e5ade915-6533-4fe3-b1bf-6702782dac1f), it can be used as a keyword in the test scripts to identify the activities. This way, the keywords can then be transformed to Java methods to be executed by the test scripts. Because Java has defined rules to the names of the methods, this id is modified in the following way by the parser: - It appends the “activity." string at the beginning (Java methods can’t start with numbers and since the id is a hexadecimal string

\[ http://www.java.com/en/ \]
\[ https://www.microsoft.com/net/default.aspx \]
\[ http://go.sap.com/index.html \]
it can happen); -The “-” is transformed in “_”. As a result, the id e5ade915-6533-4fee-b1bf-6702782dac1f will be modified to the Java method activity_e5ade915_6533_4fee_b1bf_6702782dac1f(). Given this XPDL file, it’s easy for the parser to generate the test scripts. Since a valid path in the diagram represents a flow to be tested, a test script is generated by the parser for each path in the diagram. This is done in the following way: - Following the transitions in the diagram, respecting the gateway conditions, to generate a list of activities to be executed for each path; -For each activity, generate a Java method in the Test Library; - Combine the activities for each path and generate the test scripts using the Java methods of the Test Library; The outcome will be the following: -The several test scripts that calls the Java methods corresponding to the keywords (ids of the activities) which are implemented in the Test Library; -The Test Library that contains the implementation of the Java methods referenced in the test scripts. Doing this, each Java method (keyword) needs to be implemented only once for all the test scripts, which is a great advantage in this solution. For example, the login is an activity that is present in all the paths. If this Test Library didn’t exist, it was necessary to implement it in all the test scripts.

3- Implementing the generated keywords in a Test Library through the use of a Tool Factory and a project Common Lib. The Tool Factory gives automation tool independence and the project Common Lib contains all the shared objects;

4- Executing the generated test scripts in the TestAut framework which provides several features such as scheduling, results reporting, central repository, version control, different automation tools to run, mail notifications and many others extra features;

In the following figures, figure 1 is a sequence diagram of the major steps of the solution, showing the main activities involved in each one. Figure 2 is the overview picture of the global approach. It represents the components and their relations.

B. Use Case Model to the BPM Model

The first step of this solution - and the first important part - is to translate Use Cases, Business Rules and Storyboards to BPM diagrams. These diagrams must reflect the way the application works. However, Use Cases only contain the business specification, they do not reflect the application screens and the way the developer implements the business rules. The matching between Business Rules and the application screens still needs to be done. For example, to access an operation the login must be done first, but the Use Case for that operation does not reflect this pre-requisite. It only cares about what the operation should do, not how it should be done.

A Use Case consists of a set of flows which are divided into steps. Each step can also reference Business Rules. These Business Rules contain validations, calculations or other kind of algorithms. The main flow is called the normal flow, which is the best case scenario that the main actor (habitually a user) will do. From this main flow it is possible to go to the alternative flows. The transformation of a Use Case to a BPM diagram is done in the following way: (1) Start in step one of the normal flow and identify all the test conditions for all the steps of the flow. When the normal flow is covered, the test conditions of the alternative flows should be identified; (2) For each of these test conditions, identify what happens when they are true (actions); (3) Map each of these test conditions to the application screens; (4) Start building the BPM diagram following the order of the screens, identified on the Storyboards, and doing the following: a) The actions extracted from the Use Case and the application screens are mapped to activities; b) The identified test conditions and the application decisions are mapped to exclusive gateways; c) Group the elements of the diagram according to the Use Case flows (if, for some reason, something changes in the Use Case, it is easy to see where the diagram needs to be altered, since there is a visual association); (5) Represent the jump of the normal flow to the alternative flows by link events. There are two reasons for this choice: first, to simplify the diagram with fewer lines turning it more clear and readable. The second reason is to make an
analogy with the methodology of flows identification in UP. For instance, in a test scenario that initiates in step 1 of the normal flow and then continues step by step until step 5 where it “jumps” to alternative flow 5A, with a UP methodology it becomes represented in the following way: 1.Normal, 5 -> 5A where “->” represents a jump in the flow. The link events try to reflect this approach; (6) The rules of the BPMN must be followed: a) The BPM diagram must have the start and termination events b) An exclusive gateway can’t have more than an entry. If more than one transition must be connected to the gateway, it is necessary to use an auxiliary non-conditional gateway; c) A condition associated to an exclusive gateway cannot be connected to more than one activity. In this case it is also necessary to use an auxiliary non-conditional gateway.

C. Keyword-Driven Test Cases

Technically speaking, keywords are the basic and reusable building blocks for keyword test design. Keywords define actions that drive or get information from application objects. Keywords are defined with an action-descriptive keyword name (e.g. Hotel Reservation System Start, Hotel Reservation Enter, Hotel Select, Reservation Confirmation Validate, or Hotel Reservation Close) and keyword argument names (e.g. Country, City, CheckinDate, CheckoutDate). For GUI applications each keyword is associated with one (or sometimes more) GUI windows. Each keyword argument may be associated with a window object, like a list-box or a button. Reuse of Keywords ensures rapid test development and easy maintenance.

Keyword-driven testing is a type of functional automation testing framework and its main advantages are:
- It is best suited for novice or non-technical testers;
- Enables writing tests in a more abstract manner using this approach;
- Keyword driven testing allows automation to be started earlier even before a stable build is delivered for testing;
- There is a high degree of reusability;
- It also has a few disadvantages:
  - Initial investment in developing the keywords and its related functionalities might take longer.
  - It might act as a restriction to the technically abled testers.

In this approach, the keywords are the activities of the BPM diagram which represent the actions to be done.

The next step is to build the automated test scripts using the identified keywords. This is done by translating each keyword to a Java method that will be called by the several test scripts, which are generated automatically by this solution. As stated before, keywords are defined with an action-descriptive keyword name but since in this paper the keywords are Java methods, there is the restriction as to respect the Java syntax. In other words, although in the BPM diagram point of view the keywords do have representative names, when they are “converted” to Java methods, the Java syntax rules must be followed. For that limitation, it is better to use the activities unique id for the name of the keywords, since it is always a valid name. For the Java methods (keywords) identification to be easier, Java annotations are used to associate each Java method to the name of the correspondent activity in the BPM diagram.

This could be seen as a limitation but then again, since this refers to individual BPM diagrams for each Use Case and not to activities shared by different diagrams (Use Cases) an auxiliary class (Common Lib) was used, where all the common Java methods (keywords) for the different BPM diagrams (Use Cases) were implemented. The idea of the Common Lib implementation was, by some means, to overcome this limitation. Since each path of the diagram represents a Test Case, it is easy to combine the keywords by the right order and generating a functional test script. Each Java method is then implemented manually using automated testing tools, such as Selenium or SilkTest. The scripts are then imported to the TestAut framework which will execute the Java methods using the Robot Framework.

D. Automatic Generation of Keywords

Keywords introduced in the previous section are the “heart” of this solution. To automatically generate them, the BPM diagram needs firstly to be converted in a script language that can be “parsed”.

BPM modeling tools, like bizagi6 used in this approach, allow building the BPM diagram with the set of rules and methodology described in section III-B. Besides that, it is possible to export this diagram to a XPDL file, which defines a XML schema for specifying the declarative part of a business process. A Java parser was implemented is this work to make the automatic generation of the possible keywords from the XPDL file. As explained before, each keyword corresponds to a task in the BPM diagram. From the XPDL file, each task has a unique identifier. For example: 1eb39e26-9d54-48b7-a350-5b7e7e7ccbaeb89.

The implemented Java XPDL Parser in this solution will generate automatically the following:

1- Test scripts Java classes (JUnit files), each corresponding to a Test Case, with methods (keywords) that map the different activities of the Test Case scenario. For each path in the BPM diagram it will generate a test script. In this way 100% testing coverage for the BPM diagram is guaranteed.

2- A Test Library with all the different methods (keywords) used in all test scripts.

The name generated for each method (keyword) is represented by the following rule: activity_<unique identifier>. In every method there is also a Java annotation with the real name of the activity on the original BPM diagram (figure 3). Since the name of each method it’s a unique hexadecimal number, the annotation will facilitate the identification of what the method should implement. Additionally, in the case of a runtime error, when executing the test scripts, with the annotation it is possible to know in which keyword (activity from the BPM diagram) the test fails. This will help to investigate if it is a real bug in the application under testing or if it is a modification needed to be implemented in the test scripts that came from a change in the requirements (Use Cases and Business Rules).

To summarize, this Java XPDL Parser, is the main engine and the second important part of this approach, to generate all the test scripts with all the keywords, in the right order, to cover all the testing scenarios, from a XPDL file. Reaching this point, the next step is to automate each of the keywords with the help of a

6 http://www.bizagi.com/
testing Tool Factory. Figure 3 shows an example of a test script (the “skeleton”).

Figure 3 – Automatic generated test script example.

E. Test Library and Factory Pattern

This is the third and the last important part of this approach: implementing the methods in the Test Library. As mentioned in section III-A, the methods called by the test scripts are defined and implemented in a shared Test Library. This is an important part of the solution since it reduces the implementation effort. If, for example, 10 test scripts call the same method there is only the need to implement that method once. The other imperative part of the solution is the Tool Factory, which gives automation tool independence through the use of the factory pattern [8, 9]. In this pattern a common Java interface is defined and implemented by the several supported tools. If the company wants to adopt another tool, it is only necessary to implement the common interface and all the test scripts done before will continue to work. As an example of the use of this factory, let’s suppose the following method is implemented in the Test Library:

```java
public void activity_e5ade915_6533_4fee_b1bf_6702782dac1f()
{
    login()
    t.writeById("username", "20005768354");
    t.writeById("password", "testes");
    t.clickById("confirmar");
}
```

Instead of calling instructions of the Selenium or the Silktest automation tools, the factory pattern will be used in the following way:

```java
private ToolFactory f;
private Tool t;

testCaseConstructor()
    f = new ToolFactory();
    t = f.getTool();
}

Use the interface to call the instructions:

```java
public void activity_e5ade915_6533_4fee_b1bf_6702782dac1f()
{
    t.clickById("confirmar");
}
```

This is an easy-to-maintain solution and simple to adapt to the evolutions of the application which will eventually be composed by the following Java classes:

1. The test scripts (generated by the XPDL Parser);
2. The test Tool Factory;
3. The Test Library (where the keywords referenced by the test scripts are implemented with the help of the test Tool Factory and the project Common Lib);
4. The project Common Lib.

An alternative to the project Common Lib would be the sharing of the common keywords between Use Cases. However, it is not easy to share activities amongst different BPM diagrams. This would require a manual management of the XPDL file, copying the common activities ids from one file to another. Besides, the parser would need to support version management, generating new Java methods only for the activities that have been changed. This could be a future improvement to the parser.

IV. PROOF OF CONCEPT

In this chapter, the new approach presented in this paper is tested and evaluated against a real case. A Portuguese Social Security web application called QLF (Qualificação) is used for this proof of concept and a Use Case and Business Rules were chosen to test the real applicability of this solution. The Use Case is called UC_001 – Consultar qualificações de trabalhadores de EE. It has a normal flow with 9 steps and 26 alternate flows (in average each alternate flow has 3 steps). Besides that, the Use Case references 11 different Business Rules. In total there are 11 full A4 pages describing this functionality.

The Test Cases for this Use Case were already designed, months ago, in +Tester tool. It took 2 days for an experience tester to do 57 Test Cases manually and another 2 days to automate them in Selenium.

To prove this solution, the following stages should be executed and its execution times measured:

Stage 1 - BPM Modeling: implement a BPM diagram, based on the chosen Use Case and Business Rules, by applying a set of rules and methodology explained in section III-B;
Stage 2 – Generate de XPDL file: using the BPM diagram, implemented in previous stage, as the input to a BPM modeling tool (in this case bizagi), the next step is to export this diagram to XPDL file format;
Stage 3 – Generation of Tests Scripts: Having the XPDL file from stage 2, using it as the input for the XPDL Parser, it will...
automatically generate all the test scripts for this Use Case and
the Test Library;
Stage 4 – Implementing the Test Library using the Factory
Pattern: The next step is to implement in the Test Library the
Java methods corresponding to the several activities (keywords).
This should be done using the factory pattern to get tool
independence;
Stage 5 – Running the Test Scripts: Reaching this last step, the
scripts and the Test Library need to be uploaded in TestAut and
then, run the test scripts.

A. Stage 1 – BPM Modeling

This first stage is a manual process based on the set of rules
explained in section III-B.
As mentioned previously, the BPM diagram should reflect both
the business specification (Use Cases and Business Rules) and
the interface specification (Storyboards). Besides the Use Cases
and Business Rules documentation it is also necessary to
understand how the application implements the business. This is
specified in Storyboards, which are used in the UP methodology
to model the user interface and the relation between the different
screens.

Therefore, to build BPM diagrams the Business Rules must be
mapped to application screens. Since each path in the diagram
represents a test script, the activities must be put in the right
order to obtain a properly working script.
This proof of concept was done for the Use Case introduced in
section III which specifies the business rules for an operation of
the Portal of Portuguese Social Security (PTSS). This operation
allows employers to search information about their worker’s
qualifications. To see how the BPM diagram was built, let see
how the normal flow of this Use Case was modeled.
The first thing to do is identify the test conditions. For the
normal flow the following test conditions were identified: (1)
Search by Worker data. In this case there are two search criteria:
by Portuguese Social Security Number; By Rate id (2) Search by
workplace and rate. In this search, the options available are: Date
interval; Qualification type; Address of workplace; Date interval
and rate.
Having the test conditions identified, the interface specification
must be analyzed to see if all these options for the search are
available and how they are mapped to the application screens.
The mapping of the business rules to the application screens for
this case has the following result:
1- The first screen is the login page. So, the first thing is to
select a user for the login. Due to performance reasons, the
application supports an asynchronous mechanism for users who
have more than 50 employees. This feature changes several
things in the application screens. As a result, the first thing to
do in the BPM diagram is to insert 3 activities to support this
behavior: a) Select an employer with more than 50 workers; b)
Select an employer with less than 50 workers; c) Select an
employer without workers;
2- After the login is done, the user (employer) must select the
operation in the menu;
3- Looking at the storyboard screens for that operation, it is
easily seen that it has available two tabs, which support the two
kinds of searches identified in the test conditions above: a) Tab
1: search by worker data; b) Tab 2: Search by workplace and
rate;
4- Clicking in Tab 1 it can be found the two searches identified
in the Use Case: a) Portuguese Social Security Number; b) Rate
id;
5- Clicking in Tab 2, it can be found the remaining searches
identified in the Use Case: a) Date interval; b) Qualification type;
c) Address of workplace; d) Date interval and rate;
6- For the Tab 2 (Search by workplace and rate): a) If the user
has more than 50 workers, the interaction with the user is the
following: a) There is a combo box where the user selects the
type of the file to be exported with the worker’s data. The file
type can be PDF, XLS or ODS; b) The request is sent to a queue
of asynchronous tasks to be processed. When the file is generated
it is sent by mail to the user; c) If the employer has less than 50
workers, the user has to do the following: i) Click in the search
button; ii) Select the type of the file (PDF, XLS or ODS) in a
combo box; iii) Save the exported file to the local disk; iv) In this
screen there is also a link to a page where the user can search the
admission details of a specific worker;
7- For Tab 1 (search by Worker data), does not matter if the
employer has more or less than 50 users since the search is done
for an individual worker. In this case, the screen is equal to Tab 2
with less than 50 users.
8- In the end, the users can do the logout to quit from the
application.
Each of the actions identified is then mapped to an activity task...
in the BPM diagram. The decisions map to exclusive gateways. In this case we have exclusive gateways to distinguish between the following conditions: a) The user selects Tab 1 (search by worker data) or Tab 2 (search by workplace and rate); b) The employer has more than 50 workers, less than 50 workers and no workers; c) The employer has workers or have no workers (the user with no workers is alternative flow that will be added to the diagram).

As a result, the BPM diagram for the normal flow is presented in figure 4.

As it is possible to observe, non-conditional gateways were used to respect the BPMN rules, as explained in section III-B.

To finish the diagram, the same approach should be followed for the alternative flows which will be connected to the normal flow using link gateways. It took half-day to implement in bizagi.

B. Stage 2 – Generate the XPDL file

After having modeled the BPM diagram, it is possible to export it to the XPDL format. This file is generated automatically by the BPM modeling tools and will be the input to the XPDL Parser to generate all the test scripts.

C. Stage 3 – Generations of Test Scripts

Having the XPDL file, the XPDL Parser can be executed to generate the test scripts and the Test Library. This parser was implemented from scratch to automatically generate these outputs.

The test script is a JUnit Java class [11] that calls the activities. These activities (keywords) are mapped to Java methods that will be implemented in the Test Library. For the BPM diagram above (figure 4) the XPDL Parser has generated 72 test scripts and the Test Library (LibQgenUC001.java) with the java methods corresponding to the activities.

Here is one of the 72 generated test scripts:

```java
package ScriptsQgen;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import LibQgen.LibQgenUC001;
public class Script_1 {
    private LibQgenUC001 lib;
    @Before
    public void setup() {
        lib = new LibQgenUC001();
        lib.setup();
    }
    @Test
    public void test() {
        lib.activity_6e4754c7_5099_4e20_ac53_d4db95b1286a();
        lib.activity_f1972081_2037_4d32_8379_5e3b632d8cd5();
        lib.activity_f6049773_9b48_4964_b453_40891b70a6cd();
        lib.activity_4c4c2c12_a654_41e1_abe7_18db0079fdead();
        lib.activity_e0645d43_7003_4455_bd50_5248a079ff18();
        lib.activity_7dec8054_7acd_4e46_9a89_0ae3ed3a1e();
        lib.activity_e5ade915_6533_4f70_6702782dac1f();
    }
    @After
    public void tearDown() {
        lib.tearDown();
    }
}
```

The setup() method creates an instance of the Test Library which is then used in the test() method to call the activities (keywords) covered by the test script.

D. Stage 4 – Implementing the Test Library using the Factory Pattern

The next step is to implement manually, in the Test Library, the Java methods corresponding to the several activities (keywords). This should be done using the factory pattern to get tool independence and the project Common Lib. The Tool Factory supports, at the moment, the Selenium and the SilkTest testing tools, as it can be understood in the following source code:

```java
public class ToolFactory {
    public Tool getTool() {
        // Opções possíveis para este parâmetro: "Selenium" e "Silktest"
        String tool = Testautlib.getParam("TOOL","Selenium");
        if (tool.equalsIgnoreCase("Silktest")) {
            return new SilktestWeb();
        } else {
            return new Selenium();
        }
    }
}
```

This is a very simple class which has only the purpose of choosing the automatization tool. It calls the TestAut method getParam to get the value of the parameter TOOL, where the default value is "selenium". Based on this value, it instantiates one class or the other.

The classes SilktestWeb and Selenium, called by the Tool Factory, implement the tool interface, which has the definition of the methods that should be used by the test scripts. The methods of this interface were defined by analyzing the instructions sets of the automation tools and the way they work.

From this analysis two main conclusions were taken:

1- Both tools, Selenium and SilkTest, identify the objects present in the interface, essentially by the following elements:
   - Id;
   - Name;
   - Text of the links in a web page;
   - XPath expressions, which is a query language used in the Web defined by the World Wide Web Consortium (W3C)

2- The tools have instructions to click, write, clean, and check values.
Based on these investigations, the following Java interface has been defined and it contains methods such as `clickById` and `writeByName`, which use these common functionalities. The setup and teardown methods allow the user to perform actions before the beginning and after the ending of the execution, such as initializing the tool driver.

**Tool.java - interface that hides the details of the tools**

```java
package pt.segsocial.internal.aa.weblib;
public interface Tool {
    public void setup(String url);
    public void clickById(String componentId);
    public void clickByName(String componentName);
    public void clickByXPath(String xPathStr);
    public void clickByLinkText(String linkText);
    public void writeById(String componentId, String text);
    public void writeByName(String componentName, String text);
    public void writeByXPath(String xpathStr, String text);
    public void clearById(String componentId);
    public void clearByName(String componentName);
    public void clearByXPath(String xpathStr);
    public void StringCheckById(String componentId, String str);
    public void StringCheckByName(String componentName, String str);
    public void StringCheckByXPath(String xPathStr, String str);
    public void StringCheckByLinkText(String linkText, String str);
    public void tearDown();
    public Object getToolObject();
}
```

**ToolFactory.java – Java class that allows to choose the tool**

```java
package pt.segsocial.internal.aa.weblib;
import pt.ii.testaut.lib.Testautlib;
public class ToolFactory {
    public Tool getTool() {
        String tool = Testautlib.getParam("TOOL", "Silktest");
        if (tool.equalsIgnoreCase("Silktest")) {
            return new SilktestWeb();
        } else {
            return new Selenium();
        }
    }
}
```

The Test Library generated by the XPDL Parser has following structure:

1. The constructor where is created an instance of the Tool Factory;
2. A setup method where all the necessary initializations are done before the start of the test script execution;
3. A teardown method to do all the cleanup actions at the end of the test script execution;
4. The Java methods corresponding to the several activities called by the test scripts.

**Example of a Test Library:**

```java
public class LibQgenUC001 {
    private ToolFactory f;
    private Tool t;
    private String niss, ec;
    private String password;
    
    public LibQgenUC001() {
        f = new ToolFactory();
        t = f.getTool();
    }
    
    public void setup() {
        String _url = Testautlib.getParam("URL", "https://quaapp.segsocial.pt/");
        t.setup(_url + "ptss");
    }
    
    public void tearDown() {
        t.tearDown();
    }
    
    @Login
    public void activity_f1972081_2037_4d32_8379_5e3b632d8cd5() {
        PTSSCommons.login(t, niss, password);
        t.StringCheckById(PTSSCommons.idMsgHomePage, PTSSCommons.idMsgHomePage);
    }
    
    @Operação Consultar trabalhadores
    public void activity_d830cbf7_1251_49fd_914d_1b6b5556e776() {
        niss = "25105838736";
        password = "testes";
    }
    
    @Niss < 50 Trab.
    public void activity_6e4754c7_5099_4e20_ac53_d4db95b1286a() {
        niss = "10031134622";
        password = "testes";
    }
}
```

The Test Library generated by the XPDL Parser for figure 4, had 72 test scripts and took 6 hours to implement.

**E. Stage 5 – Running the Test Scripts**

By succeeding in the previous steps it is then possible to upload both the test scripts and the Test Library in TestAut and proceed to running the test scripts. This is done in the following order:

1. Upload the test scripts in TestAut
2. Upload the Test Library
3. Associate the Test Library to the test script
4. Add the following parameters to the test script:
   - TOOL – name of the automation test tool to use (The supported tools at the moment are Selenium and Silktest)
   - URL – address of the web application to test
5. Create the Test Cases and associate the corresponding test scripts
6. In the Test Cases enter the values for the parameters TOOL and URL (Selenium or Silktest)
7. Create a test suite and add the Test Cases
8. Run the suite. This will execute all the Test Cases associated to the suite. This execution can be done in the following two ways:
   a) Launching immediately the execution
   b) Scheduling an execution for a given date/time
9. Get the report with the results of the execution, generated by TestAut using the Robot Framework
As a result of this approach, the 72 scripts have run with success in both tools supported by the factory, being very easy to change the automation tool used. It is only necessary to change TOOL parameter in Testaut to Selenium or SilkTest without the need to compile or do any changes in the code or in the test scripts.

As previously seen, stage 5 has a manual step and an automatic one. The upload of the test scripts and Test Library generated by this solution is done in a manual way in TestAut, but the execution of the test scripts is automatic.

Table 1 shows a summary of which steps are manual and which are automatic in this approach.

<table>
<thead>
<tr>
<th>Manual</th>
<th>Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1: Implementing the BPM diagram from Use Case, Business Rules and Storyboard</td>
<td>Stage 2: Exporting the XPDL file from BPM modeling tool</td>
</tr>
<tr>
<td>Stage 4: Implementing the keywords on the Test Library using the factory pattern</td>
<td>Stage 3: Generation of test scripts through the XPLD Parser</td>
</tr>
<tr>
<td>Stage 5: Upload the test scripts and the Test Library in TestAut</td>
<td>Stage 5: Execute the tests scripts in TestAut</td>
</tr>
</tbody>
</table>

F. Evaluating the Results

From the proof of concept, it is possible to compare the current / old approach used in Instituto de Informática, I.P., with the new approach shown in this paper. Table 2 presents the main conclusions of the comparison:

<table>
<thead>
<tr>
<th>Current / old approach</th>
<th>New approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harder and more time consuming:</td>
<td>More simple and less time consuming:</td>
</tr>
<tr>
<td>Manual Test Case took 2 days to design 57 Test Cases and 2 days to automate them in Selenium.</td>
<td>BPM design from Use Case and Business Rules took half-day to implement in bizagi and the 72 Test Cases (test scripts) were automatically generated. The Test Library took 6 hours to implement.</td>
</tr>
<tr>
<td>Incomplete Test Coverage:</td>
<td>Complete Test Coverage:</td>
</tr>
<tr>
<td>57 Test Cases were produced manually and 15 were missed</td>
<td>72 Test Cases (test scripts) were automatically generated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Cases always need to be designed from Use Cases, Business Rules and Storyboards</th>
<th>No Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the methodology of implementing the BPM diagram from Use Cases, Business Rules and Storyboards is applied in the analysis phase, the tester doesn’t need to do this manual task and the Test Cases are automatically generated from the BPM diagram delivered from the analysis team to the testing team</td>
<td>If the Use Case, Business Rules or the Storyboard changes, the BPM diagram needs to be updated manually, but it is an easier task to change a graphical diagram than updating text-only Test Cases and less time consuming. After updating the BPM Diagram, all the test scripts (Test Cases) are automatically generated again and only the keywords that changed need to be implemented again</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Automation Testing Tool dependence:</th>
<th>Automation Testing Tool independence:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Test Cases automation process needs to be done with a specific tool: Selenium or SilkTest and the generated scripts are tool dependent. If the tool, for example, gets discontinued and a new tool is adopted, the scripts must be recorded again.</td>
<td>The test scripts (Test Cases) generated are tool independent. If the tool changes, it is only necessary to implement the Java interface of the Test factory without the need to change the scripts.</td>
</tr>
</tbody>
</table>
V. CONCLUSION

In this paper a new approach was presented on how test automation process can be optimized and be more efficient in the phase of designing and implementing automatic Test Cases. Keyword driven test approach is not new but this method, as previously shown, demonstrates that with a new way of designing functional specifications (the input to the Test Cases) it is possible to automatically generate these keywords, which are “placed” in test scripts with all the possible combinations and guaranteeing 100% functional specification coverage from the automatic Test Cases (test scripts) generated through this solution. This method also works for the automatically generation of manual Test Cases by using the annotations, another benefit from this approach.

Regarding functional specification: Use Cases, Business and Storyboards text documents are implemented in BPM diagrams with a set of rules described in this paper. In the end, having a functional specification easier to understand, to maintain and with the greater benefit of generating the test scripts, highlights the additional advantages of this approach.

Finally, this approach demonstrates how automation tool independence can be used for tests scripts generated with this solution.

REFERENCES


