Marian JURECZKO[1], Michal MLYNARSKI[2]

Technical University of Wroclaw (1), University of Paderborn (2)

Automated acceptance testing tools for web applications using Test-Driven Development

***Abstract**. In the last years the software engineering community pays a strong interest in agile development methods. Those methods place software testing for example the Test-Driven Development method as an important task of the development process. Agile projects rely on good test automation tools. In this paper we evaluate five test automation tools for their usage in acceptance testing for web applications using Test-Driven Development.*

***Streszczenie.** Rozdział prezentuje analizę porównawczą wybranych narzędzi umożliwiających tworzenie zautomatyzowanych testów akceptacyjnych, czyli Fitnesse, Proven!, jfcUnit oraz Selenium. Badana jest możliwość stosowania tych narzędzi do testowania aplikacji internetowych wytwarzanych w procesie opierającym się o podejście programowania przez testy. Aplikacje internetowe są bardzo specyficzną grupą aplikacji. Do ich uruchomienia potrzeba serwera aplikacji. W związku z tym nie każda aplikacja nadaje się do przeprowadzania na nich testów akceptacyjnych. Dodatkowe komplikacje pojawiają się, jeżeli wymagać, aby testy powstały przed napisaniem kodu źródłowego, co ma miejsce w przypadku programowaniu przez testy. (**Zautomatyzowane testy akceptacyjne dla aplikacji internetowych w programowaniu sterowanym testami**)*

Keywords: test-driven development, test automation tools, acceptance testing, web application testing.
Słowa kluczowe: programowanie przez testy, automatyzacja testow, testy akceptacyjne, testowanie aplikacji webowych.

Introduction

Nowadays software engineering pays more attention to so-called agile methods. Those methods place automated tests as one of the important tasks in software engineering. One of the famous agile methods is test-driven development (TDD), where code is implemented according to prior created test cases. The industrial practice shows, that test automation is not a trivial task. Especially in TDD the popular test automation tools using the Capture&Replay method, where the interaction between the user and system under test (SUT) is recorded, cannot be applied because the system is not implemented at the time of test case design.

Different application types need different test automation methods. For example the tests of web applications are hard to automate because of the complex architecture and several technologies used within it. Speaking of acceptance tests, a web application has to be tested in a customer-like environment. This means, that the application has to be deployed on an application server.

In order to conduct acceptance tests in a test-driven development project, the test automation tool issue has to be clarified. The goal of this article is to evaluate five test automation tools for their usage in acceptance tests conducted with TDD. We use a GQM-like method, where tools are evaluated according to predefined goals. For each tool, we first create a test case and then implement code for a small user story example based on the Spring framework.

This article begins with the description of the evaluation process. Then we introduce our running example, selected tools and we evaluate each tool according to the evaluation process. At the end we summarize the evaluation and select the most appropriate test automation tool.

Evaluation process

As mentioned in the introduction we define a clear evaluation process for the tool comparison. For this purpose we first adjust the Goal Question Metric (GQM) introduced in [1]. We define the main goal and several subgoals. For each subgoal we define questions which have to be answered. Each question has to be answered within a predefined value range. Different as in GQM we do not use concrete metrics but predefined answer values of the questions for each subgoal.

The overall evaluation process is divided into three parts: 1) tool usage 2) question answering and 3) tool comparison. As shown in Fig. we first choose a tool and create test cases with the test automation tool. Then we implement the code according to the test case definition. After step 1) we answer the questions defined for all subgoals as described earlier. To give a better understanding of our evaluation process we first describe the TDD-like activities of step 1) and then briefly introduce our goals, questions and value ranges for step 2).

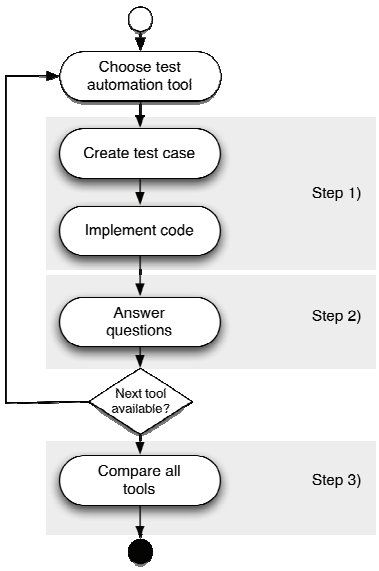

Fig. 1. Tool Evaluation Process

Test-Driven Development as described in [2] and the movement of Extreme Programming [3-4] gains high attention in the software engineering community. Different as in typical iterative development models, the test activities start before and not after the code is implemented. As test cases are used as the specification of the system, the first activity in the development process is test case design. Next, code needed to pass the created test case is being implemented. If needed a refactoring of the created code can be conducted in the last step.

According to [14] TDD can be used on two test levels. The first one is unit testing, where test cases are mostly created with tools from the xUnit family. The second and important for this article test level is acceptance testing. Based on

agile models like [5-6] we define the following activities needed to conduct TDD-like acceptance tests:

1. Select and understand a User Story for the implemented system.
2. Create new automated test cases and add to existing test suites.
3. Execute test cases. According to the red-green bar patterns [2], these test cases should fail. As the functionality needed to fulfil the selected User Story has not been implemented, the test cases cannot pass.
4. Implement the User Story functionality needed to pass the created test cases.
5. If some alternative scenarios of the User Story can be identified, then activities 2-4 should be reexecuted.

In Fig. 1 we summarize those activities with *Create test case* and *Implement code*. Knowing how the first step of our evaluation process is done, we now show what questions have to be answered in order to evaluate a test automation tool.

As mentioned earlier, our main goal is to *evaluate test automation tools for their usage in TDD-like acceptance testing*. Based on the literature [2,3,5,6] and our observations, we define the following subgoals G1-G10, their appropriate questions Q and answers A:

G1. Test first
Q: *Can the automation tool be used before or after the code is implemented?*
A: *easy, difficult, not possible*

G2. SUT type
Q: *Can the automation tool be applied for web applications?*
A: *web application, non web application*

G3. Test environment
Q: *Can the created test cases be executed in a client-like environment?*
A: *client-like, development-like*

G4. Development integration
Q: *Can the automation tool be integrated in the development environment?*
A: *possible, not possible*

G5. Test implementation
Q: *How test cases for GUI tests are created?*
A: *coding, Capture&Replay*

G6. GUI usage
Q: *Does the automation tool use id´s or position coordinates of GUI elements?*
A: *id-based, position-based*

G7. Database interaction
Q: *Can the automation tool interact with the database?*
A: *possible, possible through integration with other tools, not possible*

G8. Error checking
Q: *Can the automation tool access and check error messages?*
A: *possible, not possible*

G9. Test case readability
Q: *Are test cases readable for non software engineers?*
A: *readable, hardly readable, non-readable*

G10. Customer inclusion
Q: *Can the customer be involved in test case specification?*
A: *involved, not involved*

All introduced questions have to be answered for each evaluated tool in step 2) of the evaluation process. The order of the possible answers shown above is not random. The highest satisfaction for each subgoal is achieved by answering with the first predefined answer. For example subgoal G9 is fully satisfied if giving the *readable* answer. The subgoals can be seen as the evaluation criteria. After executing steps 1) and 2) for all selected tools, we compare them according to the values defined for each answer.

Running example

For our evaluation, we choose the wide-known web application framework Spring. The framework authors provide documentation, where a very basic inventory management system can be implemented with it [13]. For this article we use the *Change Price* user story. It can be described as follows:

A user is able to change the price of a selected product by providing the percent change value. If the price could be changed, a message with the new price should be shown. If the user provides negative or null values, then an error message should be shown.

Knowing what use case has to be implemented we briefly introduce the tools chosen for our evaluation process.

Evaluated tools

There are several test automation tools on the market. The evaluation of all available tools is not the scope of this work. We have chosen five tools, which can be categorized in open-source and proprietary. From the fist category we selected the following tools:

JFCUnit [7] is a JUnit extension for testing graphical user interfaces. Through an XML-based engine it is possible to capture and replay the needed test cases. Additionally test cases can also be implemented within XML or Java. JFCUnit is restricted to testing GUIs based on Java Swing.

Selenium [8] can be used as a browser plug-in for Capture&Replay. The Selenium API supports the manual implementation of test cases in several programming languages. Further, it supports the transformation of recorded test cases into predefined languages. This way the integration with for example JUnit is possible.

FitNesse [11] is a test automation tool based upon the Fit test framework as described in [12]. It offers all features known from Fit and enhances it with self-editable wiki pages for test case specifications. Typically test cases created with FitNesse use directly the application business logic omitting the GUI. The test cases are specified with tables representing all user action steps. Through several extensions called fixtures, it is possible to use the GUI or even the underlying database in test case specifications.

AutoIt [9] is a BASIC-like scripting language designed for automating the Windows GUI. It uses a combination of simulated keystrokes, mouse movement and window/control manipulation. Unfortunately AutoIt supports neither web applications nor Java Swing programs but on the other hand it is very popular and the application of AutoIt in the context of test-driven development may be very interesting.

For our evaluation we also selected one proprietary tool called **Proven!** [10]. It was developed by Capgemini sd&m a German software development company. The tool is based on Fit. The main goal was to deliver several features missing in other Fit-based tools like FitNesse. For example it supports the creation of test cases based on the Selenium engine. All Proven! extensions can also be used within FitNesse. Test cases are specified with tables containing five columns: the command (e.g. enter, press, select, or check), its context, connector type to be used, its arguments and optional comments. The Proven! connectors play an important role and enable the communication with the GUI, database or any predefined local or distributed system. If the standard set of connectors does not fulfil the project requirements, a new one can be easily implemented. Proven! is distributed with the Common Public License.

Evaluation results

The user story described in the second chapter was implemented according to the TDD guidelines in the conducted experiment. The user story was implemented five times. Each time another tool was used to prepare the acceptance tests.

The acceptance tests were performed separately in each of the investigated tools. We were trying to test exactly the same set of functionalities each time.

The AutoIt tests

```
;Positive Scenario
Run( "java -cp (…) springapp.gui.Main" )

If Not WinWait("MyShop", "", 10) Then
   ReportError("MyShop")
EndIf

ControlClick("MyShop", "", "IncreasePrice")
Send(10)
ControlClick("MyShop", "", "Execute")

$products =
ControlGetText("MyShop","","Products")
StringRegExp( $products, ".*Lamp 6,36.*")
If @error != 0 Then
    ReportError("Lamp 6,36")
EndIf


;Negative Scenario
If Not WinWait("MyShop", "", 10) Then
   ReportError("MyShop")
EndIf

ControlClick("MyShop", "", "IncreasePrice")
Send(10)
ControlClick("MyShop", "", "Execute")

If Not WinWait("Error", "", 10) Then
   ReportError("Error")
EndIf

$products=ControlGetText("MyShop","","Products")
StringRegExp( $products, ".*Table 64,0.*")
If @error != 0 Then
    ReportError("Table 64,0")
EndIf
```

The AutoIt tool is quite popular, but definitely it is not dedicated to test web or java applications. In order to pass the AutoIt acceptance test we had to make some simplifications. A desktop, Swing based graphical interface was developed. Unfortunately it was not enough. AutoIt is dedicated to test typical Windows applications and has some problems with Swing interfaces. Therefore, the AutoIt acceptance test is not a real test, but rather a wish scenario that shows how AutoIt can test Windows applications. AutoIt does not work well with Swing applications, thus giving a real Swing acceptance test could deceive about the real quality of AutoIt.

Writing tests before implementation with AutoIt is difficult, but possible. We can not use the 'Magic Wand' tool that reads components id´s from working application. Therefore, the components ids must be identified manually during test specification what is very time consuming. The AutoIt acceptance tests can not be executed in the customer environment because AutoIt does not work with web application at all. The tests are stored in scripts (ordinary files) and may be executed in batch mode hence they are easy to integrate with the software development environment. There is no support for database test, but there is a possibility to extend the basic set of AutoIt functionalities. Especially one may write a database extension. The AutoIt acceptance tests may test error dialogs too. The tests are not readable for a non software engineer and therefore the customer may not be involved in the test case specification.

The JFCUnit tests

```
public class PriceIncreaseTest extends
JFCTestCase {
…
@Override
protected void setUp() throws Exception {
 super.setUp();
 setHelper(new JFCTestHelper());
 SwingApp sa=SwingApp.getApplication();
 swingView=new SwingView( sa );
 sa.show(swingView);
 getComponents();
}

protected void getComponents() {
 finder=new NamedComponentFinder(JComponent.
 class,"PrcIn",false);
 mPriceIncreaseButton=(JButton)finder.find();
 finder.setName("Percentage");
 mPercentageField=(JTextField)finder.find();
 finder.setName("PriceField");
 mPriceField=(JTextField)finder.find();
 finder.setName("MasterTable");
 mMasterTable=(JTable)finder.find();
 finder.setName("ErrorLabel");
 mErrorLabel=(JLabel)finder.find();
}

public void testPositive() {
 getHelper().sendKeyAction(new KeyEventData(
this, mPercentageField, KeyEvent.VK_A,
KeyEvent.CTRL_DOWN_MASK, 100));
 getHelper().sendKeyAction(new KeyEventData(
this, mPercentageField, KeyEvent.VK_DELETE));
 getHelper().sendString(new StringEventData(
this, mPercentageField, "10"));
 getHelper().enterClickAndLeave(new
JTableMouseEventData(this,
mMasterTable, 0, 1, 1));
 getHelper().enterClickAndLeave(new
MouseEventData(this, mPriceIncreaseButton));
 assertEquals("6.36", mPriceField.getText());
}

public void testNegative() {
 getHelper().sendKeyAction(new KeyEventData(
this, mPercentageField, KeyEvent.VK_A,
KeyEvent.CTRL_DOWN_MASK, 100));
 getHelper().sendKeyAction(new KeyEventData(this,
mPercentageField, KeyEvent.VK_DELETE ));
 getHelper().sendString(new StringEventData(this,
mPercentageField, "-15" ) );
 getHelper().enterClickAndLeave(new
JTableMouseEventData(this, mMasterTable,0,1,1));
 getHelper().enterClickAndLeave(new
JTableMouseEventData(this, mMasterTable,0,2,1));
 getHelper().enterClickAndLeave(new
MouseEventData(this, mPriceIncreaseButton));
 assertEquals("Must be greater than 0!",
mErrorLabel.getText());
}
}
```

The JFCUnit tool may be used only against a peculiar type of application. Especially JFCUnit may not be used to test web applications. In order to evaluate the JFCUnit tool a desktop application with a Swing graphical interface had to be build. Therefore the application can not be tested in the customer environment. The test could be written before implementation, but it was really inconvenient because of the amount of details that had to be defined during the test specification (like names of the graphical components). The implementation was made according to the TDD principles and thus the Capture&Replay feature could not be used. JFCUnit does not support communication with database,

but as long the JFCUnit test are written in java one may use the JDBC. There are no problems with the integration in the software development environment too. The JFCUnit acceptance tests are not readable for a non software engineer (saving test in XML files will not help) and therefore the customer may not be involved in the test case specification.

The Selenium tests

```
public class PriceIncreaseTest extends
SeleneseTestCase{
 @Override
 public void setUp() throws Exception {
  setUp("http://localhost:8080/springapp",
                                  "*firefox3");
 }

public void testPossitive() throws Exception {
  selenium.open("/springapp");
  selenium.click("link=Increase Prices");
  selenium.waitForPageToLoad("30000");
  selenium.type("percentage", "10");
  selenium.select("prod", "label=Lamp");
  selenium.click("//input[@value='Execute']");
  selenium.waitForPageToLoad("30000");
  if( !selenium.isTextPresent("Lamp $6.36") )
   throw new SeleniumException("AssertError!");
  }

 public void testNegative() throws Exception {
  selenium.open("/springapp");
  selenium.click("link=Increase Prices");
  selenium.waitForPageToLoad("30000");
  selenium.type("percentage", "-15");
  selenium.select("prod", "label=Table");
  selenium.click("//input[@value='Execute']");
  selenium.waitForPageToLoad("30000");
  if( !selenium.isTextPresent(
                      "Must be higher than 0!") )
    throw new SeleniumException("Error!");
  }
}
}
```

One can test first with Selenium, but unfortunately the Capture&Replay feature may not be used in the test first approach. The Capture&Replay feature works extremely well in Selenium and probably it is the main source of high level of popularity of Selenium. Selenium works well against web applications. In fact it can test only web applications and therefore is well suited to this sort of applications. There is a possibility to start the application in customer environment too. There is no support for database test, but fortunately the test may be converted to java code and thus one can use the JDBC. The conversion into the java code simplifies the integration in the development environment as well. The tests are barely readable. The tests are written in a programming language but with very few technical details. Therefore a non software engineer may understand the tests after a short technical introduction, but writing a new test case seems to be too difficult for a non software engineer.

PriceIncreaseFixture		
product	percentage	increasePrice?
Lamp	10	6.36

New price should be 10% greater.

PriceIncreaseFixture		
product	percentage	increasePrice?
Table	-15	64.0

The price shouldn't be changed.

Fig. 2. FitNesse test tables

The FitNesse tests

```
public class PriceIncreaseFixture extends
ColumnFixture {
 public String product;
 public int percentage;

 public Double increasePrice(){
  ClassPathXmlApplicationContext ctx=new
                  ClassPathXmlApplicationContext(
                          "web/appContext.xml");
  BeanFactory bFctry=ctx
                .getAutowireCapableBeanFactory();
  ProductManager pm=(ProductManager)beanFactory
                     .getBean("productManager");
  pm.increasePrice(percentage, product);
  List<Product> products=pm.getProducts();
  Iterator<Product> itr=products.iterator();
  while( itr.hasNext() ){
     Product prod = itr.next();
     if( prod.getDescription()
                     .equalsIgnoreCase(product) )
          return new Double(prod.getPrice());
  }
  return null;
 }
}
```

Writing tests cases before implementation is easy with FitNesse. The web applications may be tested as well. But unfortunately the web application must be started in a test environment. There is an extension called JdbcFixture that supports database interactions. Using JdbcFixture requires knowledge about the SQL language what could decrease the readability level of tests for a non software engineer. The FitNesse test cases are defined in tables, but there is a java code layer too. The java code is used to execute the test cases defined in tables. The JDBC may be used in the java code layer if the JdbcFixture does not work well enough. The test tables are readable for a non software engineer and the java code layer allows constructing very sophisticated or highly specialised tests. Together it creates a powerful test framework. Fig. 2 presents FitNesse test tables. It is a typical FitNesse approach. The test examines the business logic, but does not interact with the graphical interface. Therefore the test is GUI independent (does not have to be adjusted to changes in graphical interface), but on the other hand the error message can not be tested. There are extensions (fixtures) that allow integrating FitNesse with other test tools. Especially interesting may be the WebFixture that integrates Selenium with FitNesse. Those extensions have not been discussed because using such an extension will lead to the same results as we obtained in the evaluation of one of the other test tools. For example the evaluation would be similar to the Selenium evaluation in the case of WebFixture.

Test case for Use case „Increase prices“

Testfall	Increase Prices		
	1. Positive test case		
start	Springapp	seleniumkonnektor	http://localhost:8080/springapp
press	Springapp	Increase Prices	wait=30000
press	Springapp	Prod	
select	Springapp	Prod[Lamp]	
enter	Springapp	percentage	10
press	Springapp	Execute	
contains	Springapp		Lamp $6.36
	2. Negative test case		
Start	Springapp	seleniumkonnektor	http://localhost:8080/springapp
press	Springapp	Increase Prices	wait=30000
press	Springapp	prod	
select	Springapp	prod[Table]	
enter	Springapp	percentage	-15
press	Springapp	Execute	
contains	Springapp		You have to specify a percentage higher than 0

Fig. 3. Proven! tests.

The Proven! tests

The Proven! acceptance tests were defined in a HTML table. The implementation was not necessary in order to write those tests. The test definition exposes business logic what is helpful in the TDD approach. The tests may be executed in a customer environment. The level of readability is so high that the customer (a non software engineer) can no only read and understand those tests, but may be involved in test specification as well. The Capture&Replay feature is supported, but unfortunately in the TDD approach may not be used. There is a support for database tests too. The database tests are written in a dedicated language that is not as powerful as the SQL language but then should be readable for a non database engineer. The Proven! tests may be executed in batch mode thus they are easy to integrate in the software development environment or in a continuous integration system.

Table 1. Experiment results

Tool / Goal	AutoIt	JFCUnit	Selenium	FitNesse	Proven!
G1	difficult test first approach	difficult test first approach	easy test first approach	easy test first approach	easy test first approach
G2	non web application	non web application	web application	web application	web application
G3	development-like environment	development-like environment	client-like environment	client-like environment	client-like environment
G4	possible integration	possible integration	possible integration	possible integration	possible integration
G5	coding	Capture&Replay	Capture&Replay	coding	Capture&Replay
G6	id-based	id-based	id-based	-	id-based
G7	db interaction through integration with other tool	db interaction through integration with other tool	db interaction through integration with other tool	db interaction possible	db interaction possible
G8	error checking possible	error checking possible	error checking possible	error checking not possible	error checking possible
G9	non readable	non readable	hardly readable	readable	readable
G10	customer can not be involved	customer can not be involved	customer can not be involved	customer can be involved	customer can be involved

Summary

The conducted experiment showed that there are several test tools that may be used in the TDD approach. However some of the tools may complicate the work – testing through graphical interface, when there is no implementation, is not the easiest or nicest thing. More relevant are the others evaluation criteria. JFCUnit or AutoIt may not be used to test web application. FitNesse is not the best choice when one has to execute the tests in customer environment – it could be done only with the Selenium extension (WebFixture). If the database tests are crucial, FitNesse or Proven! should be chosen. Only in Proven! the database tests are readable for customer. The level of readability of tests varies between the investigated tools. The most readable are FitNesse and Proven! tests and the less readable are AutoIt and JFCUnit tests. Selenium is somewhere in the middle.

The evaluation results are detailed in tab. 1. According to those results the best tool to test web applications that are developed according to the principles of test-driven development is Proven! Undoubtedly Proven! is a good choice, but we can imagine conditions, where another tool may work better. Therefore we hardly advise to invest some time in every software project in order to distinguish whether there is a tool that fits better.

Five different acceptance tests tools were investigated in this work. We identified several disadvantages and advantages in those tools in the field of developing web application according to the TDD principles. The work is also one of the first presentations of the Proven! tool, a new test tool that has been recently developed in the Capgemini sd&m Research department.

REFERENCES

[1] Basili B. V., Caldiera G. and Rombach H.D.: The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, pp. 528-532, 1994

[2] Beck K.: Test Driven Development by Example. *Addison-Wesley Professional*, 2002

[3] Astels, D., Miller, G., Novak, M.: Practical guide to extreme programming. *Prentice Hall PTR*, 2000

[4] Beck, K. Anders, C.: Extreme Programming Explained: Embrace Change. *Addison-Wesley Professional*, 2004

[5] Jureczko, M., Magott J.: High-level Petri net model for XP methodology. In: *Software Engineering in Progress*, Nakom, 2007

[6] Madeyski L.: Test-Driven Development - An Empirical Evaluation of Agile Practice, Springer, 2010.

[7] JFCUnit available at http://jfcunit.sourceforge.net

[8] Selenium available at http://seleniumhq.org

[9] AutoIt available at http://www.autoitscript.com/autoit3

[10] Capgemini sd&m Proven! available at http://sww.sdm.de/org/sr/produkte/proven (intranet)

[11] FitNesse available at http://fitnesse.org

[12] Mugridge R., Cunningham W.: Fit for Developing Software: Framework for Integrated Tests. *Prentice Hall PTR*, 2005

[13] SpringMVConNetBeansGlassFish available at http://wiki.netbeans.org/SpringMVConNetBeansGlassFish

[14] Koskela L.: Test Driven: Practical TDD and Acceptance TDD for Java, Manning Publications, 2007

***Authors**: Marian Jureczko M.Sc., Wroclaw University of Technology, Institute of Computer Engineering, Control and Robotics, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland E-mail: marian.jureczko@pwr.wroc.pl; Michal Mlynarski, M.Sc., University of Paderborn, Software Quality Lab, Warburger Str. 100, 33100 Paderborn, Germany E-mail: mmlynarski@s-lab.upb.de*