

IMPROVING USE OF BEHAVIOUR- DRIVEN DEVELOPMENT IN WEBSPHERE COMMERCE PROJECTS

Dawid Mateusz Bołoz

Bachelor's Thesis

May 2014

Degree Programme in Software Engineering
Technology, communication and transport





Author(s) Bołoz, Dawid, Mateusz	Type of publication Bachelor's Thesis	Date 14052014
	Pages 40	Language English
		Permission for web publication (X)
Title IMPROVING USE OF BEHAVIOUR-DRIVEN DEVELOPMENT IN WEBSHERE COMMERCE PROJECTS		
Degree Programme Software Engineering		
Tutor(s) Salmikangas, Esa		
Assigned by Descom Oy		
Abstract <p>There are many Software Developing Methodologies of which Behaviour-Driven Development (BDD) is one. Many companies use this method because of its advantages. The main goal of this project was to improve the current implementation of Cucumber tool, which reads the specification written in Gherkin (in BDD style). This change had to be made in WebSphere Commerce environment.</p> <p>Before the start of achieving this goal, the current Cucumber project needed repair because of the update of WebSphere Commerce to Feature Pack 7 version. Moreover, an update of the Cucumber libraries was necessary. The improvement focused on dividing the feature files into single scenarios and sending them separately to the real server, where they were run. After that, the server had to send the response back to the client side. When all scenarios were run, the responses from their runs are merged. At the end of this process, Cucumber Reports tool shows the results of test runs in an easily readable report.</p> <p>The next goals were to separate and change the process of automated testing in Continuous Integration. The purpose of this was to enable running tests without deploying the whole project. Jenkins is the Continuous Integration tool used in the company. These modifications were made in few steps contained in this thesis.</p> <p>The result of the thesis is a functional Cucumber improvement able to divide and send feature files to the server, run them there and create reports on the client side. The second result is Jenkins configuration, which lets run testing process on server independently from the rest of the integration process.</p>		
Keywords Java, JavaEE, WebSphere, BDD, TDD, Behavior-driven development, eCommerce, Cucumber, CI, Continuous Integration, Gherkin, Jenkins, Struts, improvement		
Miscellaneous		

Contents

Acronyms and terminology.....	3
Figures.....	4
1 Introduction.....	5
1.1 Assigner.....	5
1.2 Objective of thesis.....	5
1.3 Outline of thesis.....	6
2 Testing.....	7
2.1 Definition of testing.....	7
2.2 Motivation and advantages.....	7
2.2.1 Quality improvement.....	7
2.2.2 Verification & Validation.....	8
2.2.3 Reducing costs.....	9
2.2.4 Regression testing.....	10
3 Genres of testing.....	11
3.1 Test-driven development and automated tests.....	11
3.2 Behaviour-driven development.....	12
3.3 Living documentation.....	14
4 Continuous integration.....	15
4.1 Definition.....	15
4.2 Jenkins.....	16
5 Cucumber.....	17
5.1 How it works.....	17
5.2 How to create test – Gherkin.....	18
6 IBM WebShere Commerce.....	20

6.1	Application	20
6.2	Architecture.....	20
7	Change of implementation – improvement.....	22
7.1	Overview.....	22
7.2	Why change is needed.....	22
7.3	Improved Cucumber implementation.....	22
7.3.1	Updating libraries	22
7.3.2	Solving Classpath problem	23
7.3.3	Dividing features.....	24
7.3.4	Overriding Cucumber class.....	27
7.3.5	Getting results back.....	31
7.3.6	Merging responses.....	33
7.4	Improving CI.....	36
8	Results and conclusions.....	39
	References.....	40

Acronyms and terminology

Bug

Error, failure or fault of software that causes incorrect or unexpected result

ClassLoader

Part of Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine

Classpath

Parameter that tells the Java Virtual Machine where to look for classes and packages

Constructor

Special method in class called to create an object. It prepares the new object for use, after accepting required arguments.

Feature Pack

Collection of updates, fixes or enhancements to a software program, delivered in a single installable package

HTML

HyperText Markup Language – standard language used to create web pages

HTTP

HyperText Transfer Protocol – a foundation of data communication for the World Wide Web

Java

Computer programming language

Java EE

Java computing platform, which provides program interface for developing and running enterprise software

JSON

JavaScript Object Notation – format that uses readable text to transmit data objects

JUnit

Unit test framework for Java, very important in test-driven development

Path

Form of name of file or directory, specifies a unique location in a file system.

Struts

Framework for Java, used to present and control data

Figures

FIGURE 1. COSTS OF A BUG FIX.....	9
FIGURE 2. TEST-DRIVEN DEVELOPMENT SCHEMA.....	11
FIGURE 3. BEHAVIOUR-DRIVEN DEVELOPMENT SCHEMA.....	13
FIGURE 4. SCHEMA OF CONTINUOUS INTEGRATION	15
FIGURE 5. CUCUMBER LAYERS.....	17
FIGURE 6. DIAGRAM OF SIMPLIFIED VIEW OF WSC COMMON ARCHITECTURE	21
FIGURE 7. CUCUMBER LAYERS IN DESCOM’S IMPLEMENTATION	25
FIGURE 8. CODE RESPONSIBLE FOR EXTRACTING SINGLE SCENARIO FROM FOUND SCENARIOS.....	26
FIGURE 9. CODE RESPONSIBLE FOR EXTRACTING SINGLE SCENARIO FROM FOUND EXAMPLES IN SCENARIO OUTLINES.....	27
FIGURE 10. FRAGMENT OF OVERRIDDEN METHOD <i>RUN</i> IN CLIENTCUCUMBER IMPLEMENTATION	28
FIGURE 11. STRUTS CONFIGURATION.	29
FIGURE 12. STRUTS WORK SCHEMA	29
FIGURE 13. FRAGMENT OF <i>PERFORMEXECUTE</i> METHOD, RESPONSIBLE FOR ADDING PROPER PATH.	30
FIGURE 14. CREATING SERVERCUCUMBER OBJECT AND RUNNING TESTS.	30
FIGURE 15. GETTING RESPONSE AND SENDING BACK VIA HTTP RESPONSE	31
FIGURE 16. SCHEMA OF RUNNING TESTS ON SERVER INVOKED AT CLIENT SIDE.	32
FIGURE 17. MERGING RESPONSES PROCESS	33
FIGURE 18. PROCESS OF SENDING PARAMETERS TO SERVER	34
FIGURE 19. CREATING PATH FOR REPORT GENERATOR	34
FIGURE 20. RESULTS OF CUCUMBER’S TESTS RUNS.	35
FIGURE 21. RESULTS OF EXEMPLARY FEATURE.	36
FIGURE 22. THE SCRIPT RESPONSIBLE FOR COPYING BUILT PACKAGE WITH TESTS TO SERVER	37
FIGURE 23. THE SCRIPT RESPONSIBLE FOR UNPACKING PACKAGE WITH TESTS ON THE SERVER	37
FIGURE 24. CUCUMBER CONFIGURATION OF PUBLISHING TESTS RESULTS AUTOMATICALLY	38

Tables

TABLE 1. SETS OF FACTORS	8
--------------------------------	---

1 Introduction

1.1 Assigner

The assigner of the project was Descom Oy, a marketing and technology company providing Electronic Work Environments and Electronic Channels for Marketing Sales, including Smarter Commerce Solutions, Product Information Management, Order Management and Smarter Marketing. Moreover, it designs and delivers Server and Storage Solutions. Descom Oy has eight offices in Finland, Sweden, Denmark and Poland (*We are a new age of marketing and technology company*).

1.2 Objective of thesis

There are many Software Developing Methodologies of which Behaviour-driven development (BDD) is one. Many companies use this because of its advantages. The main reason is the fact that BDD makes it possible to find a way of communication between customers and developers.

The goal of this thesis was to change BDD implementation. Descom uses Cucumber open source software, which helps with running stories/ features written in a special language called Gherkin. Gherkin is easy readable for both customers and programmers. A default solution for testing creates separate running server or mocking objects and connections, however the company wants to run and test a real environment. In effect, an employee of the company, Diego Ballve changed the original version of Cucumber and JUnit; however there were some problems: all tests were invoking and running at once and after every single change in feature file the whole package with tests had to be deployed to server and unpacked there manually. The purpose of the thesis was to improve these changes and get over with these problems. Furthermore, an additional task was to run the tests automatically after deploying the whole project on server.

1.3 Outline of thesis

Chapters from 2 to 7 contain the theoretical background, which is necessary to understand the topic in a proper way.

Chapter 8 presents how the goal of the thesis was achieved, what was changed and why.

The final chapter presents how and when the goal of thesis was achieved.

2 Testing

2.1 Definition of testing

Software testing in general is checking if the software does what it is supposed to do and what it needs to do. This process finds defects before putting the software into use. The tests use artificial data, which is usually provided by the customer. The results of tests show errors, anomalies or information about the programs non-functional attributes (Sommerville, I. 2010). The role of software testing is to make sure that the product will be acceptable to its end users and purchasers.

According to Edsger Wybe Dijkstra (Buxton, J.N., Rabdell B. 1970, 16):

"Testing shows the presence, not the absence of bugs".

The bugs almost always are in any software. They are there because of complexity of code that humans cannot completely handle due to their limited ability to manage complexity (Pan, J. 1999).

Myers (Kaner, C., Falk, J., & Nguyen, H. O. 1999) described a simple program that contained just a loop and a few IF statements - more or less 20 lines of code. When he counted all possible paths it became evident that program had 100 trillion paths. The solution for this is not to establish program functions properly under all conditions but only establish them under specific conditions.

2.2 Motivation and advantages

2.2.1 Quality improvement

According to James Whittaker (Whittaker, J. 2012), quality is achieved by mixing development and testing, merging them and remembering that one cannot be done without the other. There is no possibility to test quality directly; however, it can be done by testing three sets of factors (See Table 1).

Table 1. Sets of factors (Pan, J. 1999)

Functionality (exterior)	Engineering (interior)	Adaptability (future)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Good testing should provide measures for all significant factors. In the typical project, the key factors are usability and maintainability. However, reliability and integrity are also relevant, because of the human presence in the project. The results of measurement these factors show the extent to which the software was produced correctly. (Pan, J. 1999)

2.2.2 Verification & Validation

The goal of verification is to check that a product meets its non-functional and functional requirements; however, the aim of validations is that a customer gets what she or he expects. Validation is important because specification sometimes does not reflect the real wishes or needs of user and customer. (Sommerville, I. 2010, 207.) Not every fault is caused by bad coding. Most of the faults are the results of mistakes made during requirements definition.

Barry Boehm sums up the difference between verification and validation in these words:

“Verification consists in checking that we are building the product right, and validation consists in checking building the right product”.

The goal of verification and validation is to provide confidence that product is good enough for its expected use. Level of this confidence relies on the current marketing environment, system purpose and of course the expectation of software users. Thus,

verification and validation are supported by testing at different stages of the software development. (Sommerville, I. 2010, 206.)

2.2.3 Reducing costs

Detecting fault before putting software into use reduces the cost of repairs and re-tests. Moreover, the use of testing in the development process increases the efficiency and detects possible areas of improvement. Testing is also a source of information, which project managers can rely on to report on progress and operations. This shows that testing software can save money (Charrett, A.-M. 2007).

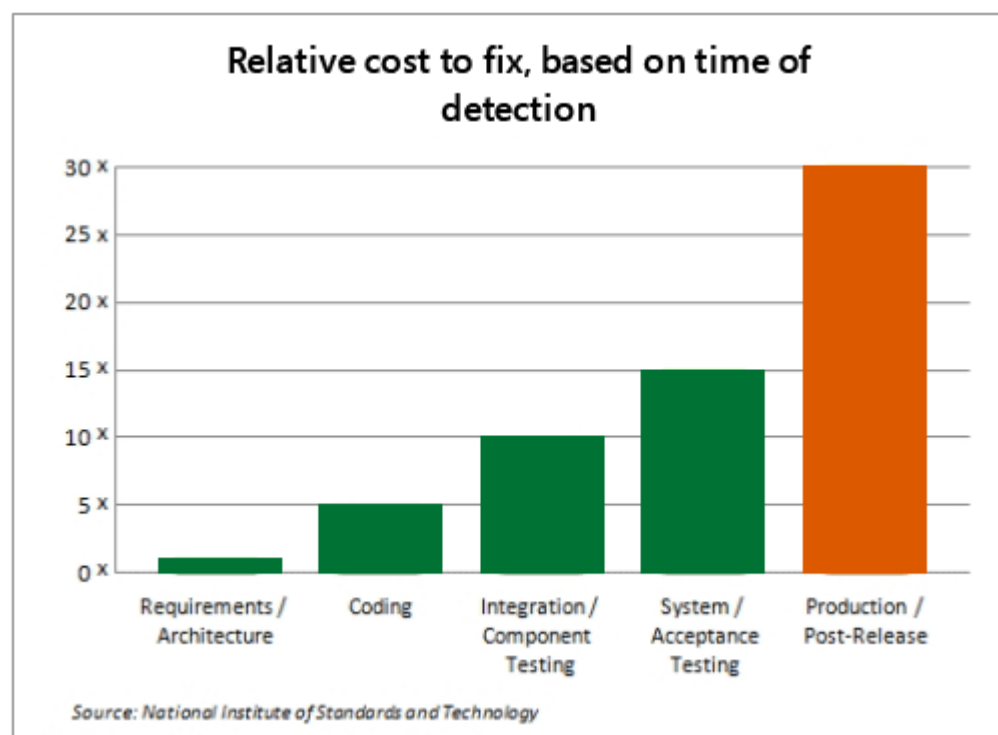


Figure 1. Costs of a bug fix (*Benefits of the SDL*)

The Figure 1 shows dependency between the cost of fixing a bug and the stage of development it is caught in. If it is caught in the Requirements stage fixing it simply costs the time of rewriting the specification. When a bug is found in the Coding stage, developer already understands the problem, and more or less knows how to fix it.

Finding the bug at the Integration level costs twice as much, because of checking many versions of code and configurations. A bug caught in the Testing stage requires time of reproduction steps, fixing the bug and verifying the fix. However, the worst scenario is if a bug is found in Production stage. In that case fixing the bug requires work and time of many people: developer, support, project manager, quality assistant and customer. (Hargraves, C. 2009.)

2.2.4 Regression testing

Regression testing should be run after making any functional improvement or repair to the program. Its goal is to check whether the change did or did not corrupt some other functionality of the software. Regression testing is achieved by rerunning some subset of the program's test cases (Myers, G. 2004). This practice is appreciated especially in projects supported for a long time. Test cases will multiply in more important and problematic areas, which ensures their proper working.

3 Genres of testing

3.1 Test-driven development and automated tests

Through many years of testing, programmers have noticed that they are writing code and after that they are modifying it to meet their requirements. This is not good practice, therefore, they have come up with an idea to write the test before writing code. The idea of writing tests before the development starts is helping to eliminate many misunderstandings before they influence codebase (Wynne, M. & Hellsøy A. 2012). That solution creates a new concept of software development process.

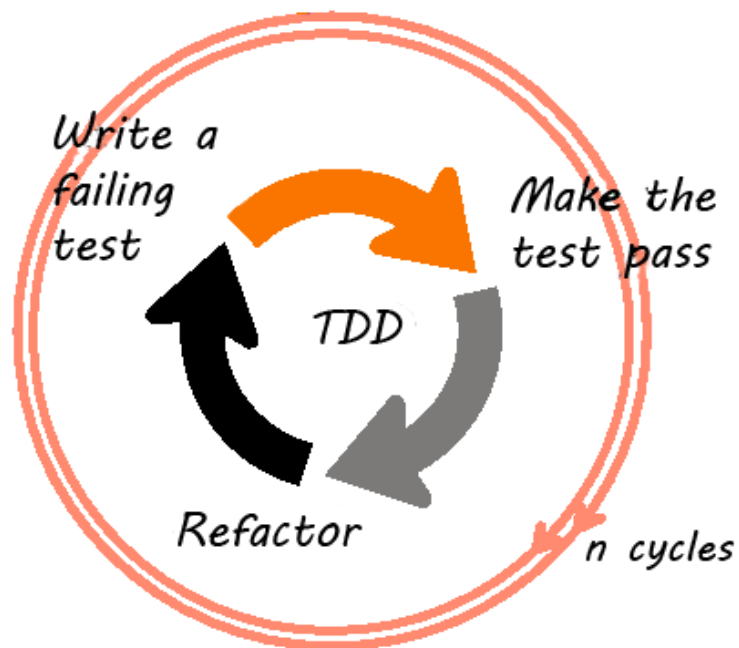


Figure 2. Test-driven development schema

Figure 2 shows fundamental TDD process. In this process rather necessary is using automated type of tests. Automated testing is use of special software, which is not connected with tested software, to control effectiveness of test by comparison output

values to expected values. This means that after tests execution separate software reports about passing or failing. The first step in Diagram is to identify the increment of functionality, which is base for the second step – writing a test. The first run of new test should fail, because there was not any implementation. In the implementation step only this amount of code should be written to pass this test (Sommerville, I. 2010). This prevents writing unused functionality. The cycle from the diagram in Figure 2 has to be repeated until the specification does not contain any more functionality, which has not been implemented yet. At that time, programmers are sure that whole implemented functionality meets all requirements, and the software works as it should. This is provided and secured when all tests have passed.

A very important attribute of TDD test is that it must be isolated. Each test cannot interact with others, because it creates situations when one test fail causes failures of hundreds of others. It looks like there is a pail of defects with features described by tests, however, in reality there is no major issue, only a small fix in the first failed test (Beck, K. 2000).

Using Test-Driven Development helps programmers to clarify their ideas of what every fragment of code is responsible for. When every segment of code has associated at least one test, then it is quite sure that all code in software has been executed, and defects are discovered early in the development process (Sommerville, I. 2010). The most important benefit of using test-driven development is the facility to regression testing. Before adding any new functionality to code, all existing tests must run successfully.

3.2 Behaviour-driven development

Behaviour-driven development (BDD) is a methodology of software development based on test-driven development. The main advantage of using BDD is writing tests as examples that everyone in the team can read. Thus, business stakeholders are giving feedback to programmers that they understood an idea or not, before the process of writing code has even started (Wynne, M. & Hellsøy A. 2012).

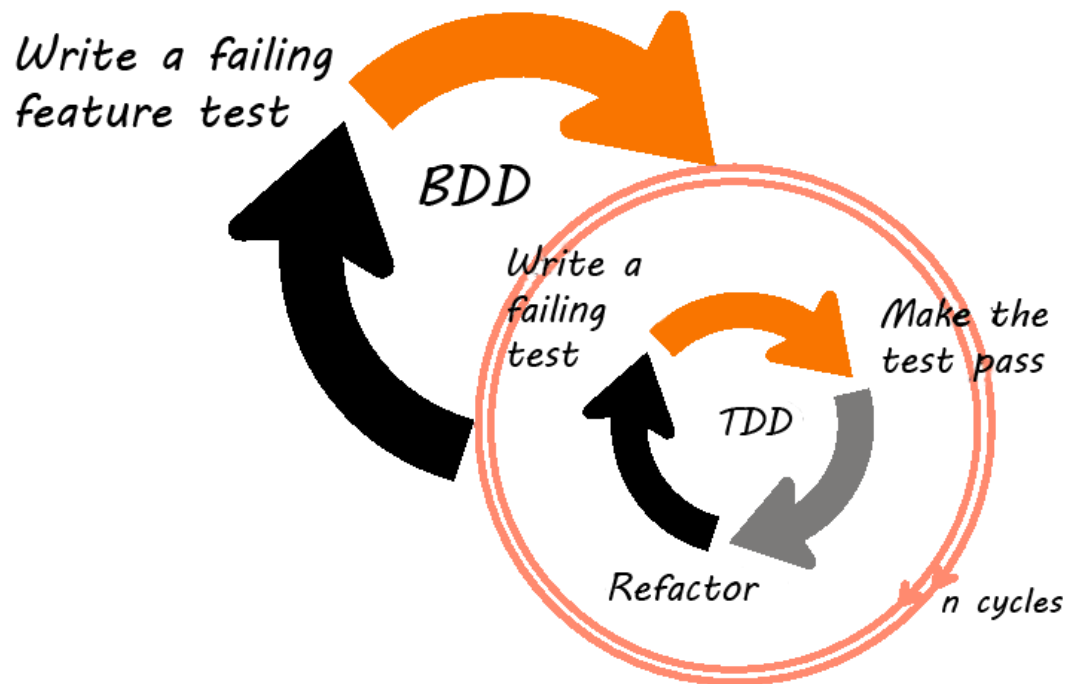


Figure 3. Behaviour-driven development schema

Behaviour-driven development was invented also because of confusions and questions from the programmers. They wanted to know where to start, what to test, what not to test, how to understand why a test fails, what to call tests, and how much to test in one go. The first solution was to replace the test method names with sentences, describing what a method is responsible for. Moreover, replacing “test” with “behaviour” answers some earlier questions. A sentence describing behaviour is a relatively good name for a test. Behaviour described in a single sentence should be maximum amount to test in one go (North, D. 2010).

As Eric Evans (Evans, E. 2003) wrote in his book “Domain Driven Design”, communication between stakeholders and programmers is very low-quality:

“A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language... Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand.”

Stories provided by stakeholders should be very brief. Firstly, good idea was to write them in plain-English description like “Do this, then that, then verify this” (Kniberg, H. 2007). So Dan North (North, D. 2010) came up with an idea of defining that kind of ubiquitous language, to give both sides of the linguistic divide a possibility of meeting. This language should not be too artificial to be understood by an analyst and it had to keep some structure to break the story into smaller fragments and to automate them. Thus, they started describing requirements in terms of scenarios, which look like this (North, D. 2010):

Given some initial context (the givens),

When an event occurs,

Then ensure some outcomes.”

This invention helps team members to decide what behaviour is needed to be implemented next. They also learn how to describe that behaviour in a language that everyone can understand.

3.3 Living documentation

Scenarios created by using a language, which is understood by everyone in the team lets people visualize the software before it has been built. Tests written in this language become more than just tests; they are executable specifications. (Wynne, M. & Hellsøy A. 2012.) They can be used as a base for development and also as a document to get clarification from stakeholders. If changes are necessary, they have to be done in only one place. (Spec. by example, 2011.) It means that documentation is not something written once and is going out of date. It is a living thing, which pictures current state of project. Using this kind of documentation saves money by keeping every part of project synchronized. Furthermore, it builds trust in the team, because everyone has one version of truth (Wynne, M. & Hellsøy A. 2012).

4 Continuous integration

4.1 Definition

Continuous integration (CI) is a software development practice. It relies on integrating and building the project many times a day, every time, when task is completed. Often project integration keeps code up to date, downsizes chances of conflicts and reduces the cost of integration. (Beck, K. 2000, 47-58.) Just after successful project build, all of the tests should be run. It has all advantages of regression testing plus creates a natural end to a development episode. Figure 4 shows the cycle of CI.



Figure 4. Schema of Continuous integration

During developing a programmer cannot ignore relationship of the changes she or he makes to the changes anyone in the team happens to be making. This leads to having code out of any control. By often integrating programmers become aware where the

collisions are: in the definition of classes or methods, and by running the tests they become aware of semantic collisions (Beck, K. 2000, 77).

Practicing CI in software development dramatically reduces the risk of the project. In few hours programmers see e.g. if they have different ideas about the appearance of a piece of code. They will not be in situation that they spend days fixing a bug, which was created few weeks earlier. At the end, when it comes to creating final project, there is no big problem, because every programmer have been doing this every day for whole project development time.

4.2 Jenkins

Jenkins is a CI tool written in Java, which makes the process of integration of the project faster and easier. In many companies the process of building the project means that a pile of scripts has to be run manually and it is taking hours of precious time for get this done. Jenkins gives a possibility to automate this process. By defining series of tasks in Jenkins it is possible to create a separate environment just for building process. After build, Jenkins can test built project and give report about success or fail. Simultaneously it can also be configured to send messages via e-mail, Google Talk, IRC, Skype to different teams or team members about changes. For example, if build process failed it should inform programmers that they broke something. (Bołt, W. 2011.) Moreover, Jenkins offers scheduling builds.

5 Cucumber

5.1 How it works

Cucumber is a tool, which reads specifications written in BDD style, checks them for scenarios to test and runs those scenarios in the tested environment. Each test case in Cucumber contains several steps and is called a scenario. The steps instruct Cucumber what to do. The scenarios are grouped into features. The specification stored in ".feature" files must be written due to Gherkin syntax for Cucumber to be able to read it (Wynne, M. & Hellsøy A. 2012, 7). Keywords: *Feature*, *Scenario*, *Given*, *When*, *Then* are the base of this syntax. The sentences located after those words are treated in case *Feature* and *Scenario* as descriptions, but those after *Given*, *When* and *Then* are names of methods which should contain the steps definitions in the language of programming used in a project.

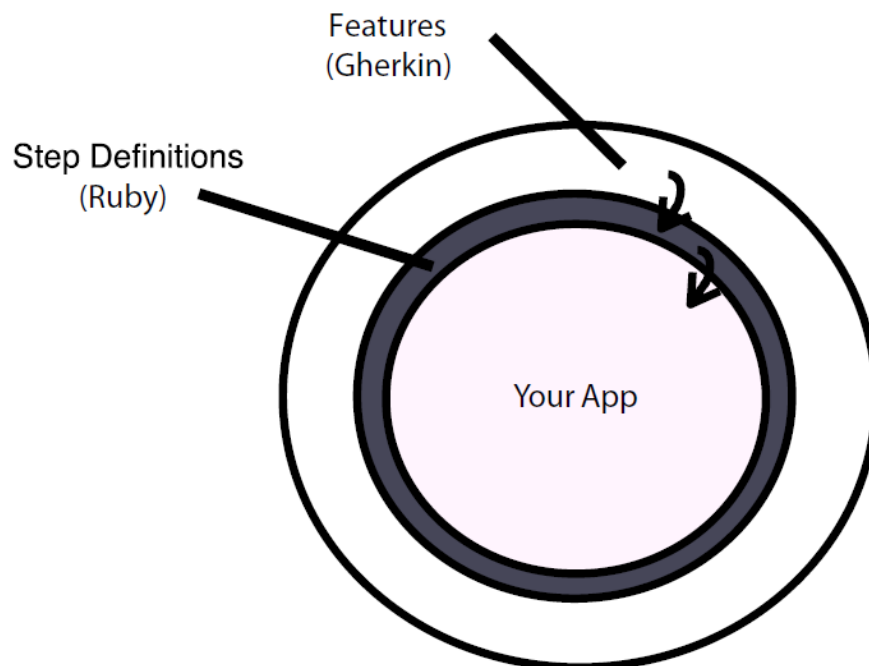


Figure 5. Cucumber layers (Wynne, M. & Hellsøy A. 2012, 15)

The Figure 5 shows the main layers of building software with Cucumber. It has to be started with writing Gherkin feature file, which contains scenarios with steps. The steps call steps definitions, which have a connection between features and building software.

A good practice is to first write the feature file with a scenario and the related step definitions and then run this test even when it is obvious that it is going to fail, because the code to test does not exist yet. This practice makes programmers sure that they have a fully functional test, before they start working on a solution. In coding it is excellent to remember to do the minimum useful work to pass test. It sounds lazy, but in fact, this is some kind of discipline. When tests are run after any sensible change, every mistake is found very quickly and Cucumber gives plenty feedback and presents the status of progress. (Wynne, M. & Hellsøy A. 2012, 24.)

5.2 How to create test – Gherkin

Gherkin has a lightweight structure for writing documentation which describes the behaviour of the wanted software and can be understood by stakeholders, programmers and by Cucumber. Although Gherkin's main goal is to be readable by everyone, it is still a programming language. (Wynne, M. & Hellsøy A. 2012, 13.) Below is an example:

```
Feature: Some terse yet descriptive text of what is desired
  Textual description of the business value of this feature
  Business rules that govern the scope of the feature
  Any additional information that will make the feature easier to
understand
```

```
  Scenario: Some determinable business situation
    Given some precondition
    And some other precondition
    When some action by the actor
    And some other action
    And yet another action
    Then some testable outcome is achieved
    And something else we can check happens too
```

```
  Scenario: A different situation
```

```
    ...
```

An interesting attribute of Gherkin is that syntax exists in many spoken languages, for now it is 37 (*Gherkin*). So it does not matter which language stakeholders or users speak, it does not lose its functionality. Each feature should contain from 5 to 20 scenarios and each of them uses different examples, to fully test the behaviour of this feature in different circumstances. There are few conventions: one feature file contains only one feature, each scenario must make sense and to be able to run separately.

6 IBM WebSphere Commerce

6.1 Application

IBM WebSphere Commerce (WSC) is a software platform framework for industry where the buying and selling of products or services is handled over electronic systems, mainly over the Internet. For business users it provides easy in use tools to manage entire shop, by using which they can create and manage precision marketing campaigns, promotions, catalogue, and trading across all sales channel. WSC is a customizable, scalable, and high availability solution that is built to leverage open standards. (*WebSphere Commerce product overview.*) It offers the ability to do business with businesses, directly with customers, indirectly through channel partners or all of them at the same time. This framework is built on Java – Java EE platform and is using open standards such as Web Services.

6.2 Architecture

WSC is built from many parts, they, combined together, create network of dependencies and relationships. The easiest way of general understanding is to follow diagram shown on Figure 6.

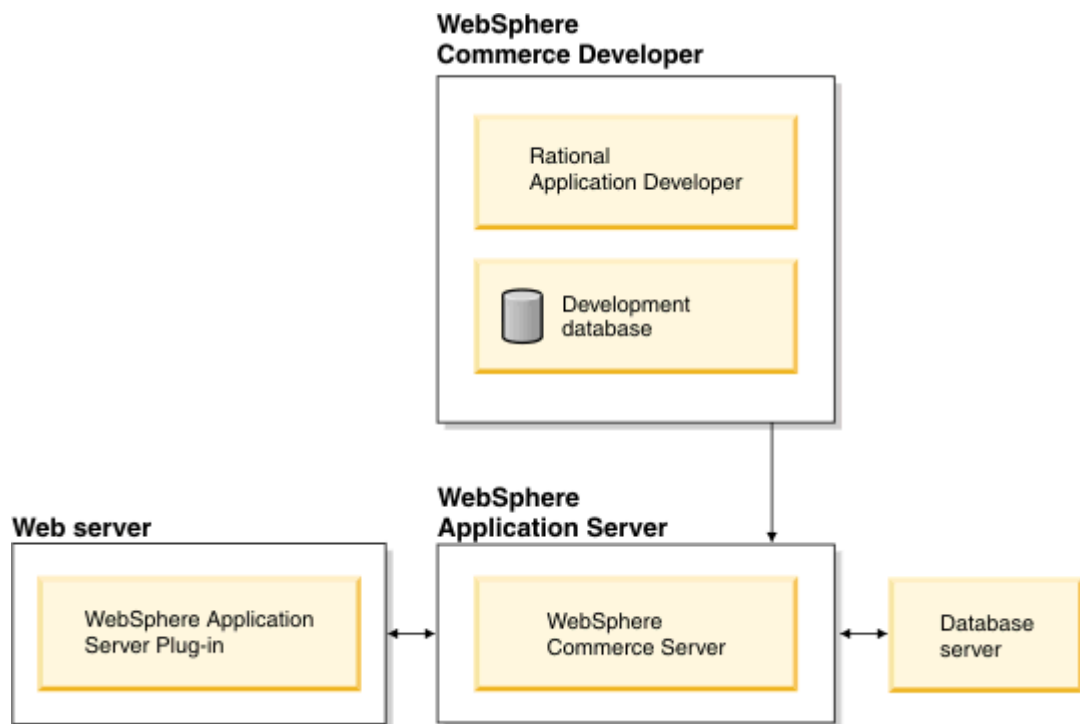


Figure 6. Diagram of simplified view of WSC common architecture

The contact with WCS starts with receiving HTTP request to Web server. It uses WebSphere Application Server Plug-in to properly and fast manage connections with WebSphere Application Server (WAS). WebSphere Commerce Server runs inside WAS, by which it has access to many features offered by application server. Database server contains and store most of application's data, including data of products and customers. Extensions can be made by modifying or extending the code for the WCS. Rational Application Developer helps in:

- creating and customizing storefront assets such as JSP and HTML pages,
- creating and modifying business logic in Java,
- creating and modifying access beans and EJB entity beans,
- testing code and storefront assets,
- creating and modifying Web services.

WebSphere Commerce Developer environment has its own development database. Programmers can use their preferred database tools to perform database modifications. (*WebSphere Commerce common architecture.*)

7 Change of implementation – improvement

7.1 Overview

The Descom Company has already been using the BDD. The programmers wrote tests in Gherkin and the steps definitions in Java. Therefore, change in Cucumber testing process had to be made, because they wanted to run tests on a real server, not to mock connections or objects. The project has already been maintained as a separate part of the files and projects of WebCommerce, thus, joining it is always available, and even removing it does not affect the operation of the project.

7.2 Why change is needed

Change is needed because the current way of invoking tests on server is ineffective. After every change in feature files, the whole package has to be deployed and unzipped on a server. There is no control on which tests run on a server, and also a summary report from server is sent only after execution of all features. So in case of a too long process, a too large report file or corruption of report file, the process must be repeated.

The second needed improvement affects CI configuration. The company wanted to change the way of invoking the automated testing process after successful build. That change was to made possible changing text of the tests and deploy only this package to the server.

7.3 Improved Cucumber implementation

7.3.1 Updating libraries

The work on this task starts with tracing actual changes provided more or less one and half year ago. From that time a new version of Cucumber and libraries connected to it came up. Some of them had only little changes, however, there were few new libraries created by extracting them from Cucumber core library. Library *cucumber-core* changed from 1.1.1 version to 1.1.6 and from extracting parts responsible for Gherkin language *gherkin* library was created. New libraries were also *cucumber-html*

and *cucumber-jvm-deps*, the presence of which is now necessary. In *cucumber-junit* library there were no changes. Packing all of them to one jar was a good idea for their easier management.

Updating libraries was connected with cosmetic changes in code. For example, the options of Cucumber were no longer nested class in Cucumber class. It is a separate class now, which has really helped in carrying out subsequent changes.

A list of modifications between versions 1.1.1 and 1.1.6 (*History*):

1. New features:
 - a. Generating *stepdef* metadata with *--dotcucumber*.
 - b. Showing class name of exceptions in the HTML formatter.
 - c. Deferring table header and column mappings.
 - d. Upgrade to Gherkin 2.8.0
2. Bug fixes:
 - a. Escape exceptions in HTML formatter
 - b. Retry when feature element returns "failed"
 - c. Rerun formatter output was not including failed scenario outline examples
3. Changed features:
 - a. Breaking long lines in output.
 - b. Slight changes in JSON formatter output.

7.3.2 Solving Classpath problem

Before starting with the actual improvement, solving Classpath problem was necessary. Feature pack 7 and new version of WebCommerce showed up the problem of not finding feature files in the project deployed on the server. It caused the author of the thesis plenty of struggles. After tracing code and debugging both sides – clients and servers, it occurred that Cucumber libraries are using Classpath taken by the reflection from test class and it was working well. The problem starts in *junit* libraries, where Classpath is taken from *CurrentThread* object, which in default did not contain paths to every test. A solution to that was to add a path of package with tests

to the `CurrentThread` object and pass it along also to Cucumber invoker in a way that every part had the same paths. At first thought it should have resolved the problem, but even if it was having path to package with tests it still was not able to find *.feature* files. The next idea, how to repair this bug was to unzip the jars on server side to get single files. It turned out that this operation was ended with success. The tests worked using updated Cucumber and new Feature Pack.

7.3.3 Dividing features

So far Descom's Cucumber implementation works very ineffective. The tests run on client side were doing all processes of finding files there, checking their structure and validating it, which was unnecessary. After finishing these processes, information to start the same tests was sent via HTTP, passing only name of the test file. Thus, on the server side, all of the preparing processes were run again and were looking for files there and validating them. The results of these processes on client side were not saved and the tests there were not invoked. From this behaviour one simple solution concludes – to get rid of one run of these preparing processes. Conversation with supervisor helped with finding a better solution. It was an idea of cutting the preparing processes in half, which sounded good. The processes of finding and validating of feature files are still on the client side. After that, the founded feature files are divided to simple scenarios and sent over http to server.

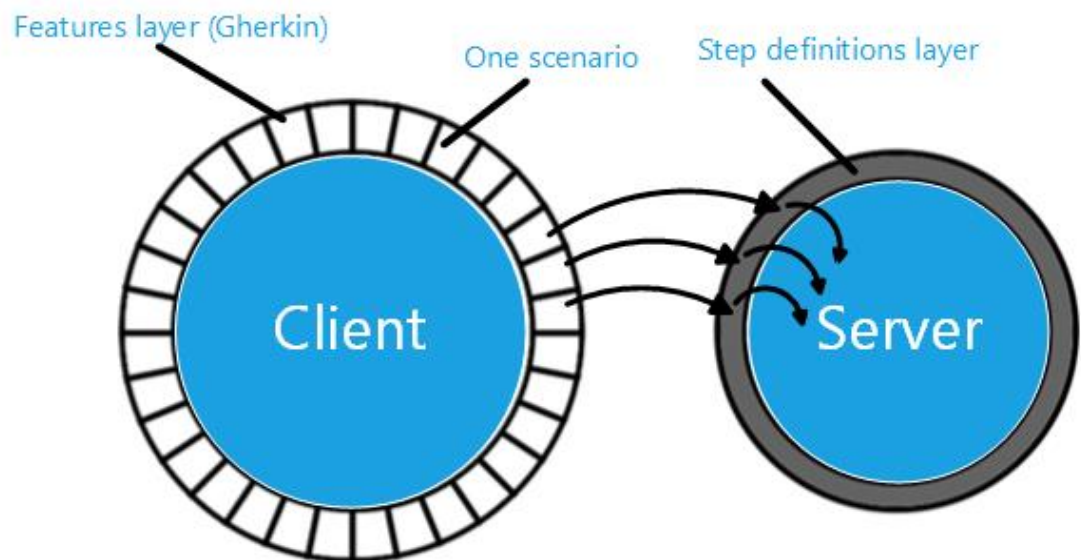


Figure 7. Cucumber layers in Descom's implementation

On the server side, the preparing processes are not started over like in the earlier implementation. They are continuing from the place where they were broken on the client side and they are looking for steps definitions files. The vision of this is shown on Figure 7.

The first thing to do was to find the code responsible for reading feature files. This process was strictly combined with building Feature objects. Therefore, not to break the structure of Cucumber algorithm, the best thing was to extend FeatureBuilder by overriding its *run* method. The overridden method replaced running found features and invoking the rest of the normal Cucumber processes on client side by dividing the features into single scenarios and sending them to server.

```

65 Queue<String> stepsToSend = new LinkedList<String>();
66
67 // getting background
68 List<Step> backgroundSteps = ((CucumberScenario) cucumberTagStatement)
69     .getCucumberBackground().getSteps();
70 if (!backgroundSteps.isEmpty()) {
71     stepsToSend.add("Background:");
72 }
73 for (Step step : backgroundSteps) {
74     stepsToSend.add(step.getKeyword() + step.getName());
75 }
76
77 // Scenario: ...
78 stepsToSend.add(cucumberTagStatement.getVisualName());
79
80 // Steps
81 List<Step> steps = cucumberTagStatement.getSteps();
82 for (Step step : steps) {
83     for (Argument outlinearg : step.getOutlineArgs()) {
84     }
85     for (Comment comment : step.getComments()) {
86     }
87     stepsToSend.add(step.getKeyword() + step.getName());
88 }
89 this.featureSenders.add(new FeatureSender(testName, "Feature: "
90     + cucumberFeature.getGherkinFeature().getName(), stepsToSend));

```

Figure 8. Code responsible for extracting single scenario from found scenarios

The process of building scenarios was replaced by building resources to send. The first part of both code examples starts with copying background steps from feature to each scenario (see Figure 8, lines 67-75 and Figure 9, lines 93-102). The next part extracts scenario steps names with keywords. The last part is slightly different because text extracted in the code showed in Figure 8 (lines 89-90) is saved in a new FeatureSender only once, however, in code in Figure 9 (lines 111-133) new FeatureSender object is created for every example. FeatureSender is a custom class created by the author of the thesis to handle process of storing extracted scenarios and theirs later sending process.

```

92 Queue<String> stepsToSend = new LinkedList<String>();
93 // getting background
94 List<Step> backgroundSteps = ((CucumberScenarioOutline) cucumberTagStatement)
95     .getCucumberExamplesList().get(0).createExampleScenarios().get(0)
96     .getCucumberBackground().getSteps();
97 if (!backgroundSteps.isEmpty()) {
98     stepsToSend.add("Background:");
99 }
100 for (Step step : backgroundSteps) {
101     stepsToSend.add(step.getKeyword() + step.getName());
102 }
103 // Scenario Outline: ...
104 stepsToSend.add(cucumberTagStatement.getVisualName());
105
106 List<Step> emptySteps = cucumberTagStatement.getSteps();
107 // adding real steps
108 for (Step step : emptySteps) {
109     stepsToSend.add(step.getKeyword() + step.getName());
110 }
111 stepsToSend.add("Examples:");
112 // getting examples rows
113 List<ExamplesTableRow> cucumberExamples2 = ((CucumberScenarioOutline) cucumberTagStatement)
114     .getCucumberExamplesList().get(0).getExamples().getRows();
115 // adding headers to every test as a next step
116 List<String> headerCells = cucumberExamples2.get(0).getCells();
117 StringBuilder sb = new StringBuilder("|");
118 for (String cell : headerCells) {
119     sb.append(cell + "|");
120 }
121 stepsToSend.add(sb.toString());
122 for (int i = 1; i < cucumberExamples2.size(); i++) {
123     List<String> valueCells = cucumberExamples2.get(i).getCells();
124     sb = new StringBuilder("|");
125     for (String cell : valueCells)
126         sb.append(cell + "|");
127     Queue<String> stepsWithExample = new LinkedList<String>(stepsToSend);
128     stepsWithExample.add(sb.toString());
129
130     this.featureSenders.add(new FeatureSender(this.testName, "Feature: "
131         + cucumberFeature.getGherkinFeature().getName(), stepsWithExample));
132 }
133 }

```

Figure 9. Code responsible for extracting single scenario from found examples in scenario outlines

7.3.4 Overriding Cucumber class

The next thing to do in this task was to handle the invoking process. The preparing processes are invoking by default in the constructor of the Cucumber class. Because of breaking them, which was mentioned earlier, the Cucumber class must have two implementations – one for client side and one for server.

At the client side, the most of the changes has already been made in building FeatureSender's process. Thus, only work there was to override *run* method in the ClientCucumber class. The first thing was to remove code responsible for running

tests. Against it, in this place can be write code, which sends found features to the server (see Figure 10).

```
92 @Override
93 public void run(RunNotifier notifier) {
94
95     // against running tests here, they are sending to the server
96     List<FeatureSender> featureSenders = new ArrayList<FeatureSender>();
97     for (EntireFeatureRunner child : children) {
98         featureSenders.addAll(child.getFeatureSenders());
99     }
100    // sending features
101    for (FeatureSender featureSender : featureSenders) {
102        featureSender.sendScenario();
103    }
```

Figure 10. Fragment of overridden method *run* in ClientCucumber implementation

After sending each scenario the client side waits for a response, which should contain the results of running tests on server, which were formatted to JSON. This formatting is provided by the Gherkin library. For each sent scenario, the client side receives one response. The process of merging responses is described in 7.3.6 chapter.

Sending process uses the HTTP connection. Connection is set between the client side and a webpage on the server. Proper Struts configuration tells the ActionServlet that when the incoming request points to that webpage it should invoke an action method in the CommerceTestRunnerCmd class (see Figure 11).

```

16<struts-config>
17  <!-- Global Forwards -->
18  <global-forwards>
19    <forward className='com.ibm.commerce.struts.EActionForward'
20      name='CommerceTestRunner' path='/tools/dev/CommerceTestRunner.jsp'>
21      <set-property property='properties' value='storeDir=no' />
22    </forward>
23  </global-forwards>
24  <!-- Action Mappings -->
25  <action-mappings type='com.ibm.commerce.struts.EActionMapping'>
26    <action parameter='com.descom.commerce.cucumber.commands.CommerceTestRunnerCmd'
27      path='/CommerceTestRunner' type='com.ibm.commerce.struts.BaseAction' />
28  </action-mappings>
29</struts-config>

```

Figure 11. Struts configuration.

Figure 12 shows schema of process of handling HTTP request at the server side. Proper action is determined in struts configuration, later invoked. Created response is sent to client through struts and JSP file.

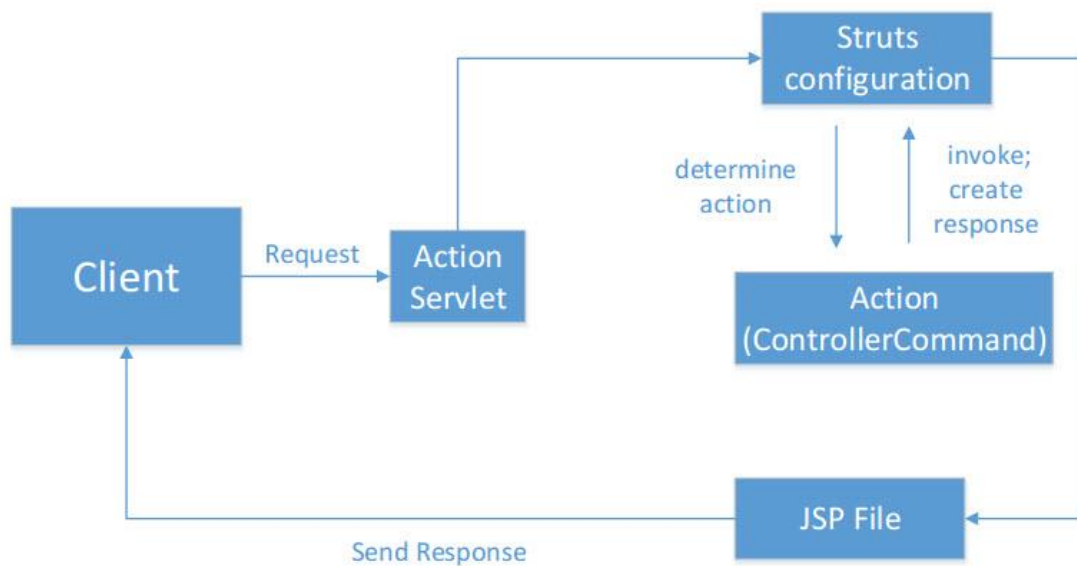


Figure 12. Struts work schema

The best thing of using this is that everyone from every place can run tests on server by requesting this webpage and giving good arguments in proper format. The name of the action method, which was mentioned earlier, in this project, is called *performExecute*. Inside this method, arguments passed via http are validated.

Moreover, strictly here problem with Classpath described in chapter 8.3.2 is resolved, by adding path of steps definitions (directory where the package with tests is unzipped on the server) to currentThread's Classloader (see Figure 13).

```

67 ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();
68 URLClassLoader classLoader = new URLClassLoader(
69     new URL[] { new URL("file:///tests/stepsDefinitions/")}, contextClassLoader);
70
71 try {
72     // setting proper classloader with all needed classpaths
73     Thread.currentThread().setContextClassLoader(classLoader);

```

Figure 13. Fragment of *performExecute* method, responsible for adding proper path.

The next action in this method is to create an object of the ServerCucumber class – custom implementation of the Cucumber class. The ServerCucumber's constructor needs the same class loader, which was changed earlier. It also needs text of scenario, and name of the feature (see Figure 14). These last two things are taken from HTTP request.

```

76 Class clazz = Class.forName(className, true, classLoader);
77 ServerCucumber cucumber = new ServerCucumber(clazz, classLoader, scenario, featureName);
78 RunNotifier rn = new RunNotifier();
79 cucumber.run(rn);

```

Figure 14. Creating ServerCucumber object and running tests.

If the scenario content is already in ServerCucumber's constructor, the process of finding feature files can be skipped. In the next part of development, it was necessary to put the passed scenario in place where preparing processes have been broken at the client side. Overriding FeatureBuilder class was essential for having this done.

Most of this class stayed the same. Only method *parse* had to be changed to receive resource as a text, not as a Resource object. Similarly, the content of this method also had to be changed. These modifications completely solved the issue of invoking tests on the server. Replacing resources in this class caused the fact that modification of any more implementation was not necessary.

After creating ServerCucumber object, next action was to run test. This was invoked in next lines of CommerceTestRunnerCmd's *performExecute* method (see Figure 14, line 79).

7.3.5 Getting results back

The last issue to do at server side was to get the test results back to the client side. After debugging and tracing code paths it became evident that writing a new implementation of the formatter is crucial. The default one was very integrated with other classes that it was impossible to change it or replace without vast interference in Cucumber's libraries code. Unfortunately, it had to be left. The implementation of new JSONResponseFormatter was based on existing JSONFormatter class from Gherkin's library. The changes were only made in *close* method, which is called just before the end of testing. At that time all results are ready to be printed out or saved to a file, however, in the new implementation, they are assigned to a class variable, which is available from ServerCucumber class.

```
79 cucumber.run(rn);
80
81 json = cucumber.getJSONResponse();
82
83     ...
84
87 ServletOutputStream outputStream = response.getResponse().getOutputStream();
88 outputStream.write(json.getBytes(Charset.forName("UTF-8")));
89 outputStream.close();
90
```

Figure 15. Getting response and sending back via http response

The *performExecute*'s next action is getting the JSON response from ServerCucumber object and passing it as Http Response via still open HTTP connection. Properly formatted response from running one scenario should come back to client side. The process of running one scenario is shown on Figure 16 and it is marked by blue arrows. It is invoked for every scenario found at the client side at the start of whole testing.

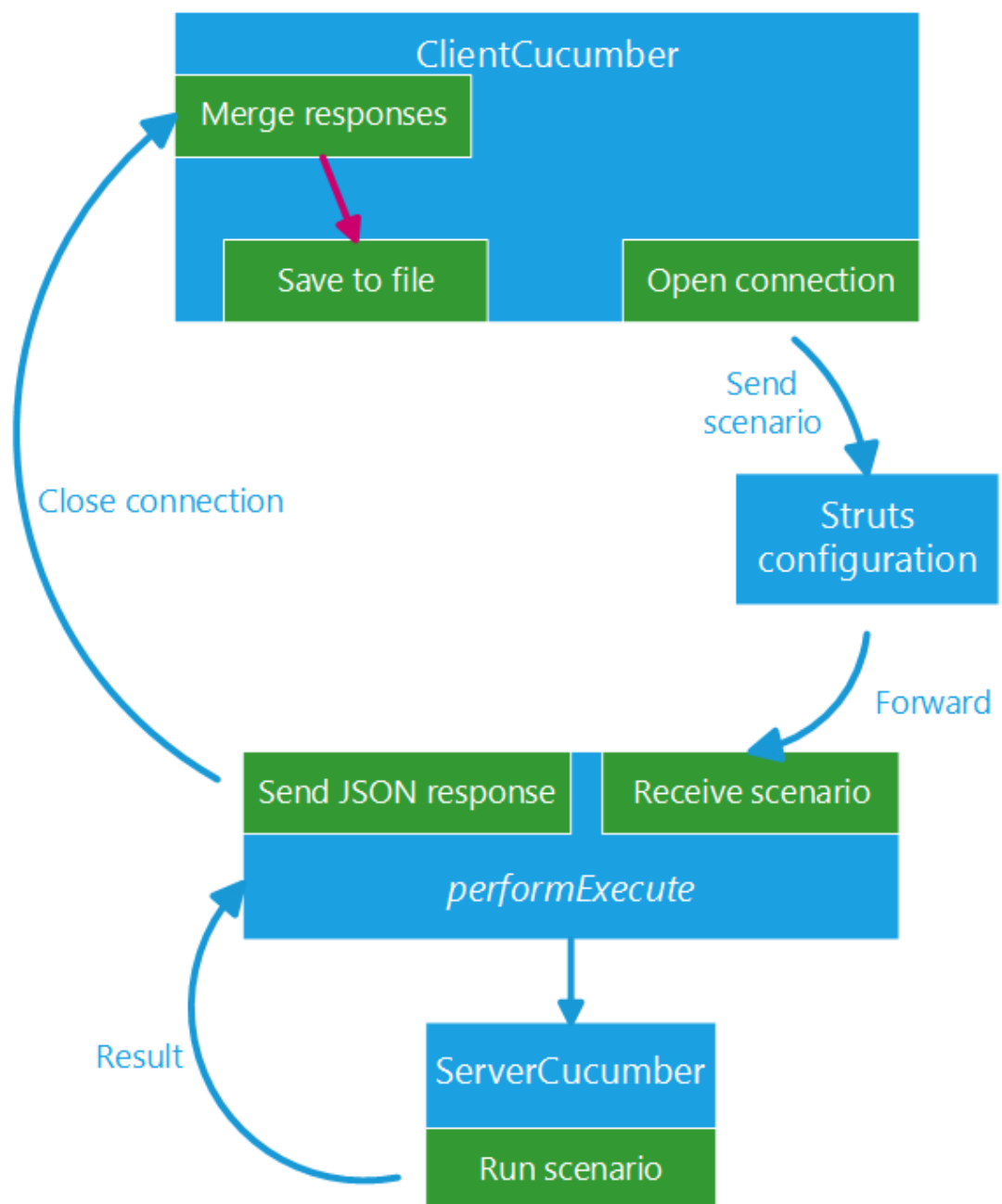


Figure 16. Schema of running tests on server invoked at client side.

7.3.6 Merging responses

When all scenarios from every found feature were already sent to server and the responses came back, it was necessary to merge them. This was achieved by comparing the features attributes to merge every scenario to the right features, and every example to the right scenario outline.

```
1109 //merging responses
1110 for (FeatureSender featureSender : featureSenders) {
1111
1112     List<Map<String, Object>> fromJson = gson.fromJson(featureSender.getJsonResponse(),
1113         featureMaps.getClass());
1114
1115     String keyword = (String) fromJson.get(0).get("keyword");
1116     String name = (String) fromJson.get(0).get("name");
1117     String id = (String) fromJson.get(0).get("id");
1118     String description = (String) fromJson.get(0).get("description");
1119
1120     int theSameFeature = -1;
1121     boolean isAdded = false;
1122     for (int i = 0; i < featureMaps.size(); i++) {
1123         Map<String, Object> map = featureMaps.get(i);
1124
1125         if (map.get("keyword").equals(keyword) && map.get("name").equals(name)
1126             && map.get("id").equals(id)
1127             && map.get("description").equals(description)) {
1128             ((List<Map<String, Object>>) map.get("elements"))
1129                 .addAll(
1130                     (List<Map<String, Object>>)fromJson.get(0).get("elements"));
1131             System.out.println("added");
1132             isAdded = true;
1133             break;
1134         }
1135     }
1136     if (!isAdded) {
1137         featureMaps.addAll(fromJson);
1138     }
1139 }
1140 }
```

Figure 17. Merging responses process

To proper merging it was crucial to set good attributes values. Setting the same name of the feature for every containing scenario caused sending also name of the feature as a parameter via http request (see Figure 18) in *sendScenario* method of *FeatureSender* class.

```

44 List<NameValuePair> params = new ArrayList<NameValuePair>();
45 params.add(new BasicNameValuePair("className", className));
46 params.add(new BasicNameValuePair("featureName", featureName));
47 params.add(new BasicNameValuePair("scenario", sb.toString()));
48
--
53 url = new URL("https://" + serverName + "/CommerceTestRunner?"
54     + getQuery(params));
55 HttpURLConnection connection = (HttpURLConnection) url.openConnection();
--

```

Figure 18. Process of sending parameters to server

The received name of the feature is passed further to the ServerCucumber class. In the ServerCucumber's constructor this name is added to the path of test file located at the server. The effect of this process - path (uri) is passed to *parse* method of the ServerFeatureBuilder. (See Figure 19)

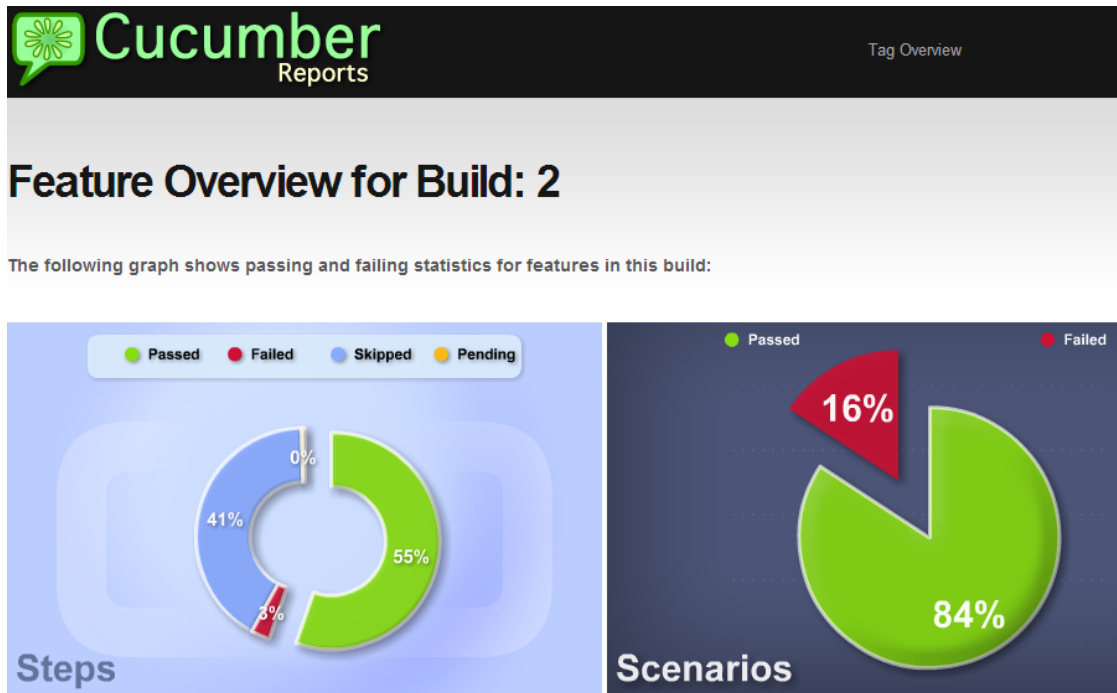
```

67
68 // creating uri for report generator
69 String uri = clazz.getName().replace(".", "\\");
70 uri = uri.substring(0, uri.lastIndexOf("\\")+1)
71     + featureName.trim() + ".feature";
72
73 // changing resource to passed string, (not file now)
74 List<CucumberFeature> cucumberFeatures = new ArrayList<CucumberFeature>();
75 ServerFeatureBuilder builder = new ServerFeatureBuilder(cucumberFeatures);
76 builder.parse(gherkin, uri, runtimeOptions.getFilters());
77

```

Figure 19. Creating path for report generator

Setting this attribute helps in merging and does not cause problems in generating a report. The process of generating a report is provided by Cucumber Reports tool, which uses properly formatted JSON response to show the results of Cucumber's tests runs in an easy and beautiful way (see Figures 20 and 21).



Feature Statistics

Feature	Scenarios	Steps	Passed	Failed	Skipped	Pending	Duration	Status
Simple test to run on server	10	26	25	0	0	1	492 ms	passed
Simple extension test to run on server	28	191	95	6	90	0	146 ms	failed
2	38	217	120	6	90	1	638 ms	Totals

Figure 20. Results of Cucumber's tests runs.

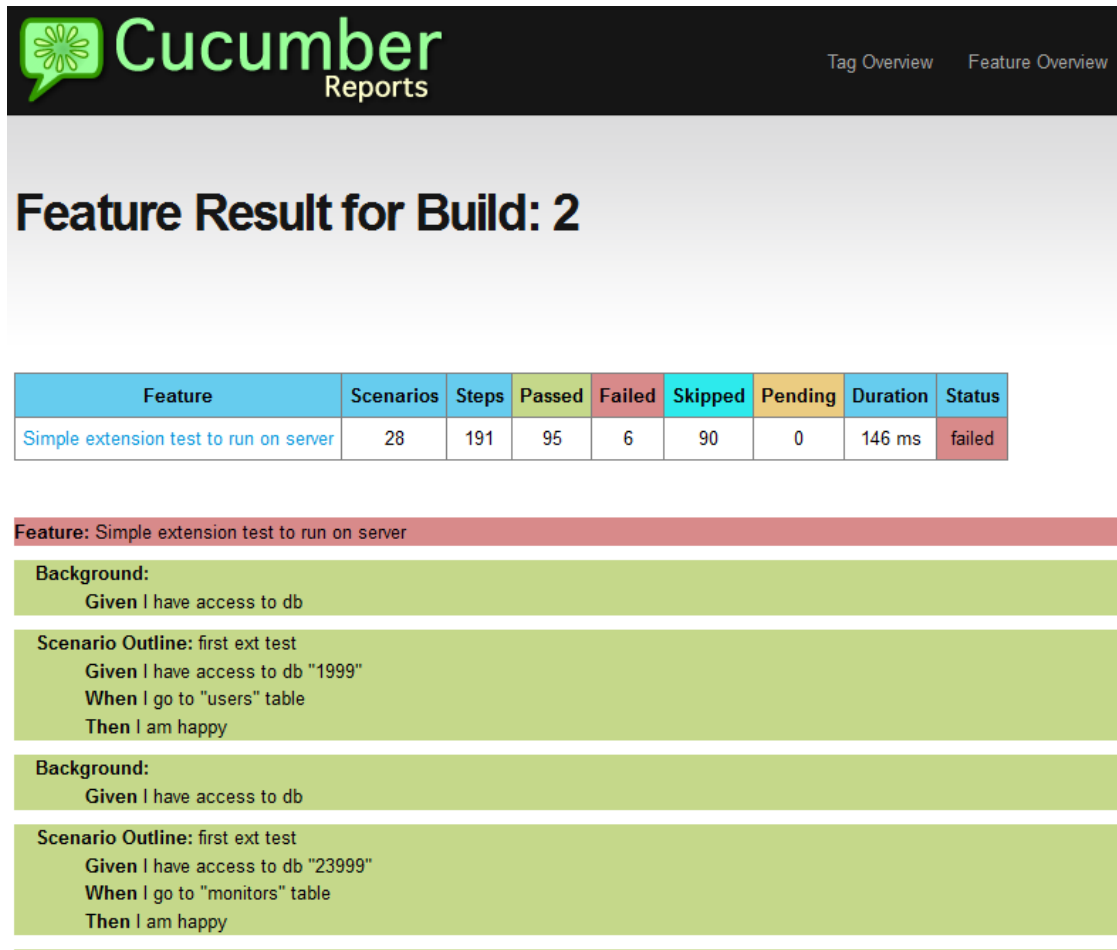


Figure 21. Results of exemplary feature.

7.4 Improving CI

The modifications of Cucumber works, thus, new changes had to be deployed to the server. This process has already been automated and Jenkins was configured to handle this job. However, the company wanted to separate package with tests from WebCommerce and Cucumber projects. The reason of that was to have possibility to update tests on server and run it independent from other projects.

The easiest thing to do was to exclude the package with tests from the existing Jenkins job. The process of creating a new job and invoking series of tasks was more difficult. Jenkins invokes Ant scripts, where it has instructions how to build, deploy, run tests, etc. The script of secure copying is shown on Figure 22.

```
55<target name="deploy" description="Deploying test jar to server" depends="jar">
56<scp todir="${user}@${server}:${target.dir}"
57   file ="${packagename}"
58   password="${password}"
59   trust="yes"
60   sftp="true"/>
61</target>
```


Figure 22. Script responsible for copying built package with tests to server

Because of Classpath problem (resolved in chapter 7.3.2) the deployed package needed to be unpacked. It had to be done using a secure connection. The Figure 23 shows how it was achieved.

```
69<target name="unzip" description="Unzipping test package">
70  <sshexec host="${server}"
71    username="${user}"
72    password="${password}"
73    trust="yes"
74    command="unzip ${target.dir}/${packagename} ${unzipped.dir} "/>
75</target>
76
```

Figure 23. The script responsible for unpacking package with tests on the server

The last thing was to write the script, which runs tests and another one, which invokes the Cucumber tool to create a report. After setting the path to the created report (Figure 24) Jenkins shows the results automatically.



Publish cucumber results as a report

Json Reports Path
The path relative to the workspace of the json reports generated by cucumber-jvm e.g. target - leave empty to let the plugin find them automatically

Plugin Url Path
The path to the jenkins user content url e.g. http://host:port[/jenkins/]plugin - leave empty if jenkins url root is host:port

Skipped Steps Fail the Build
Tick this if you want skipped steps to cause the build to fail

Pending Steps Fail the Build
Tick this if you want pending steps to cause the build to fail

Turn Off Flash Charts
Tick this if you want to use javascript charts instead of flash charts

Figure 24. Cucumber configuration of publishing tests results automatically

8 Results and conclusions

Writing this thesis was very difficult for me. So far, every project which I made was from the start to the end written by me. Thus, I know how to achieve goals more or less at start of my work. Therefore, in this project I had to handle plenty of legacy code. I traced code not only written by a Descom employee; however, also by the author of the Cucumber libraries and it was exhausting for me. From this I have learned that I should comment every single block of code for better understanding for later improvements.

The result of my work is very helpful for the company. This improvement saves plenty of time and at the same time money. It makes using BDD easier. I get rid of unnecessary code that was lengthening the process of integration in CI.

The most difficult thing I have noticed is the change in the thinking of people. The programmers still wants to create code before writing tests. They cannot get used to this change of development process. They are thinking that their way is easier, and they are right. Therefore, they are not creating the customer vision of the software but their own vision. Sometimes they aim well and achieve the goal; however, sometimes they spend a great deal of time because of some misunderstanding. When the code from start is built in the correct way from the start and the changes are not that significant, the code is easier to maintain.

I achieved the goal of improvement of projects, however, now it is necessary to also improve the programmers thinking. They need some courses and of course practice. Within time using BDD will become a great deal easier.

References

- Beck, K. 2000. *Extreme Programming Explained*. Massachusetts: Addison-Wesley.
- Benefits of the SDL*. Page on Microsoft website. Accessed 9 May 2014. Retrieved from <http://www.microsoft.com/security/sdl/about/benefits.aspx>.
- Bołt, W. 2011. *Ciągła integracja - Pan Jenkins przybywa na ratunek*. Accessed on 8 May 2014. Retrieved from <http://www.trzeciakawa.pl/?p=181>.
- Buxton, J.N., Raddell B. 1970. *Software Engineering Techniques*. Report on a conference sponsored by the NATO Science Committee. 16.
- Charrett, A.-M. 2007. *Benefits of software testing*. Accessed on 29 April 2014. Retrieved from <http://mavericktester.com/archive/benefits-of-software-testing/>.
- Evans, E. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Massachusetts: Addison-Wesley.
- Gherkin*. Page on Cucumber repository on GitHub. Accessed on 3 May 2014. Retrieved from <https://github.com/cucumber/cucumber/wiki/Gherkin>.
- Gojko, A. 2011. *Specification by Example: How Successful Teams Deliver the Right Software*. Greenwich: Manning Publications.
- Hargraves, C. 2009. *The Cost of Bugs*. Accessed on 7 May 2014. Retrieved from http://tech.lds.org/index.php?option=com_content&view=article&id=238:the-cost-of-bugs&catid=1:miscellaneous.
- History*. Page on Cucumber repository on GitHub. Accessed on 3 May 2014. Retrieved from <https://github.com/cucumber/cucumber/blob/master/History.md>.
- Kaner, C., Falk, J., & Nguyen, H. O. 1999. *Testing Computer Software, 2nd Ed*. New York: Wiley.
- Kniberg, H. 2007. *Scrum and XP from the Trenches, How we do Scrum*. Unites States of America: C4Media.
- Myers, G. 2004. *The Art of Software Testing*. New York: Wiley.
- North, D. 2010. *Introducing BDD*. Accessed on 2 May 2014. Retrieved from
- Pan, J. 1999. *Software Testing*, Carnegie Mellon University. Spring 1999. 2-6
- Sommerville, I. 2010. *Software engineering*. Harlow: Pearson Education.
- We are a new age of marketing and technology company*. Page on the Descom Oy' s website. Accessed on 2 May 2014. Retrieved from <http://www.descom.fi/>.
- WebSphere Commerce common architecture. Page on IBM Information Center website. Accessed on 8 May 2014. Retrieved from <http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/index.jsp?topic=%2Fcom.ibm.commerce.developer.doc%2Fconcepts%2Fcsdsoftwarecomp.htm>.
- WebSphere Commerce product overview*. Page on IBM Information Center website. Accessed on 8 May 2014. Retrieved from <http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/index.jsp?topic=%2Fcom.ibm.commerce.developer.doc%2Fconcepts%2Fcsdsoftwarecomp.htm>.
- Whittaker, J. 2012. *How Google Tests Software*. Massachusetts: Addison-Wesley.
- Wynne, M. & Hellsøy A. 2012. *The Cucumber Book. Behaviour-Driven Development for Testers and Developers*. Dallas: The Pragmatic Bookshelf.