On Requirements for Acceptance Testing Automation Tools in Behavior Driven Software Development

Evgeny Pyshkin
Dept. of Computer Systems and Software Engineering
Saint-Petersburg State Polytechnical University
Saint-Petersburg, Russia
pyshkin@ftk.spbstu.ru

Maxim Mozgovoy
Information Systems Division
University of Aizu
Aizu-Wakamatsu, Japan
mozgovoy@u-aizu.ac.jp

Mikhail Glukhikh
Dept. of Computer Systems and Software Engineering
Saint-Petersburg State Polytechnical University
Saint-Petersburg, Russia
glukhikh@kspt.ftk.spbstu.ru

Abstract— **We study approaches to use behavior-driven development (BDD) model while testing software. The paper's focus is on challenges of BDD in regards to the automation for mapping use cases written in narrative manner to unit tests. We analyze existing toolkits aimed to facilitate integration of the BDD libraries (e.g. JBehave) with development environments. We define requirement check list for further analysis of the BDD and IDEs integration solutions.**

Keywords- software; testing; unit testing; test driven development; behaviour driven development; programming; testing automation

Abstract in Russian— **в статье обсуждается модель разработки и тестирования программного обеспечения на основе поведенческого описания. Работа посвящена проблемам, возникающим при автоматизации генерации тестовых классов на основе вариантов использования приложения, написанных на естественном языке. Рассмотрены имеющиеся средства, реализующие концепции поведенческого тестирования, рассмотрены подходы к написанию приемочных тестов и к автоматизации их преобразования в тестовые классы на примере инфраструктуры JBehave, определен контрольный список требований к средствам автоматизации определения тестов в рамках интеграции поведенческих сценариев и средств разработки программного обеспечения.**

Ключевые слова- программное обеспечение: модульное тестирование; приемочные тесты; разработка через тестирование; тестирование через определение поведения; программирование; автоматизация тестирования.

I. INTRODUCTION

Software testing is one of the activities aimed at fixing software defects as early as possible so to improve product quality factors. Particularly, unit tests are ones that being closer to the design stage allow developers to find bugs while writing code to fit functional requirements. Developing testing concepts and methods is the essential part of software engineering theory and practice. However, one should differentiate the use of term "testing" when it indeed means "software testing techniques" from the use when it designates developing practice. It is exactly the case of test-driven development (TDD): effectively we don't write tests in the strict sense of the software testing, although we use some unit testing techniques. As Kent Beck brilliantly noticed, "Hold on there — I never said that test-first was a testing technique. In fact, if I remember correctly, I explicitly stated that it wasn't" [4]. For this reason North preferred to use the term "behavior" not only when he introduced a behavior-driven development (BDD) approach, but even in regards to the TDD itself emphasizing software product behavior aspects over testing [7].

As well as test-first techniques, the BDD approach is maturing, and there are still debates about definitions, application objects, characteristics, usage recommendations and effects [3, 5]. If we think about unit testing technique, even for a basic term of unit there is discussion on what exactly constitutes one [5]. In this paper we remain aside from such debates.

A. From TDD to BDD

Acceptance testing is an important activity in software production regardless of the development lifecycle model used in a software project [2]. Problems of acceptance testing virtually only slightly depend on the software subject domain. Every software design team is anxious about the quality of the software product and about the how the system meets customers' business need and users' expectations.

Acceptance tests are often less formal and therefore it's more difficult to formalize and automate them. Behavior driven development (BDD) is one of agile practices dealing with unit tests; however, it allows mapping the acceptance tests to the language-level test classes. In BDD, user-side test cases are being defined in form of so-called user stories, or storytests that represent the scenarios understandable by software users and customers. Thus the BDD creates a kind of a communication framework that allows the developers to rediscover the customer context better in the process of software design and testing. Hence testing is considered here to be a sort of cooperative work: the customers propose user side-test scenarios, while the testers write stories and map them to the source code constructions. It contrasts with the test-first strategy that rather helps developers to communicate clearly

with their teammates, not with the stakeholders (or, in Cunningham's words, "developers use tests to communicate with other developers" [8]). If we follow BDD, we virtually cannot prove that we are able to cover all equivalence classes, or the whole group of test cases, but this is exactly the same story as with the "test-first" TDD approach, where the term "testing" refers to the use of unit testing-style procedures rather than to the testing technique.

The BDD gives the way to formalize the stakeholders' side use cases in form of executable, readable contracts mapped to the unit tests similar to those represented by unit testing frameworks like JUnit [1-3].

B. BDD as a Developing Approach

BDD is still a developing approach, so there are many open questions regarding its usage in the software design practice and BDD tools integration with the developing environments as well. It isn't novel understanding that software tools became important factor not only in the software development but also in evaluation of design methods and concepts. The success of TDD is partially based on the support of xUnit framework implementations for various developing environments such as NetBeans, Eclipse, Visual Studio, etc. Janzen and Saiedian note that TDD and related technologies (inspired with Extreme Programming and agile development methods) may persist even if the parent technologies fade in popularity [6].

Among other issues, in the following sections we try to define some essential problems of the BDD design, usage and applications.

II. BDD COMMUNICATION SCHEMA ACCORDING TO [2], THREE PRINCIPAL ROLES HAVE TO BE CONSIDERED WHILE ORGANIZING THE PROCESS IN BDD:

- *the Customer*, who identifies user stories on the base of customer's understanding of the domain. Ideally, the customer-side engineer writes acceptance test stories in natural or quasi-natural language ("as English as possible");
- *the QA Engineer*, who does a big deal of reviewing acceptance tests, suggests new scenarios, finds problems (e.g. stories that conflict with each other or with requirements specification), transfers acceptance scenarios to the developing team in form of unit tests, and communicates both with *the Customer* and *the Developer* to define tests at the source code level;
- *the Developer*, who implements the system so it fits the requirements and passes acceptance tests.

Figure 1 shows basic steps of the acceptance test conversions as they are implemented in JBehave, one well known BDD framework [3, 10]. First, test stories are defined in quasi natural language, and it is joint work of the customer and the QA engineer. After that the unit test skeleton are constructed either manually (in most products) or with some automation. As in TDD (the ideal case), at this phase some pending steps may occur, if the code is not implemented yet.

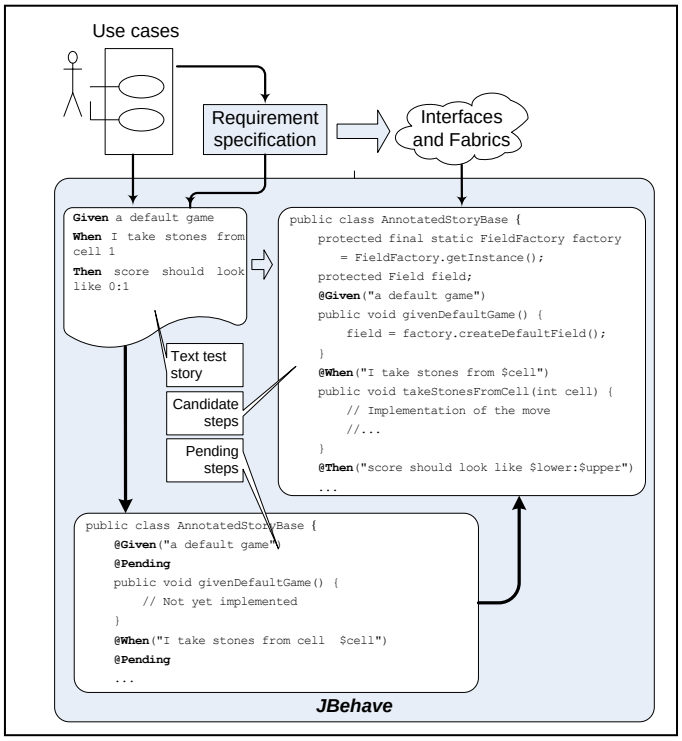

Figure 1. Conversion acceptance test to JUnit tests in JBehave

The unit tests (which they called the candidate steps) are then configured to be run by the JUnit test runners (see Figure 2).

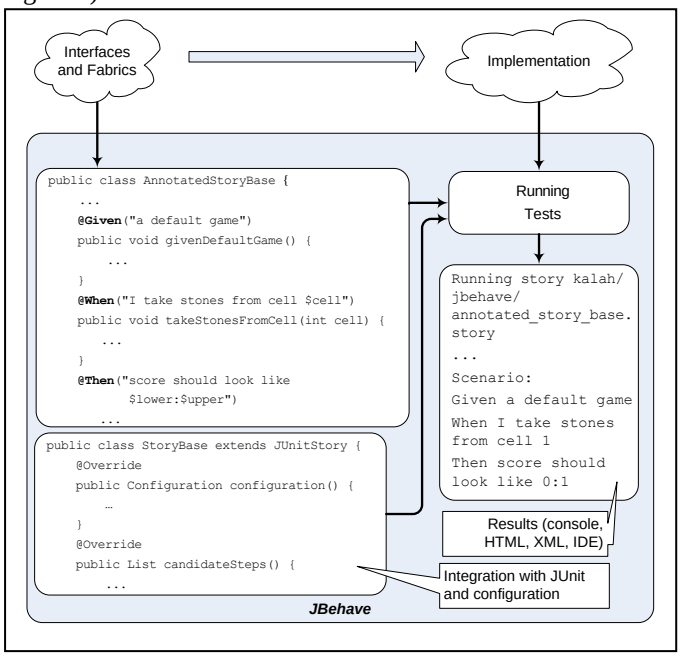

Figure 2. Running JUnit tests in JBehave

In the above illustrations we used an example logical game Kalah designed for our book on software testing. We consider logical game software to be a good example of writing test stories understandable by majority of eventual readers. In our case, we can illustrate how are text stories are being mapped to the unit test.

According to the game rules, in the initial position shown in Figure 3 the player holding the lower row of cells containing stones may, for example, take stones from the cell number 1. As a result the stones are being distributed sequentially to cells from 2 to 6 and the last stone goes to the player's (right side) Kalah giving him again the turn (see Figure 4). The example test scenarios describing the move and its consequences may be as follows:

```
Given a default game
When I take stones from cell 1
Then score should look like 0:1
Then should be lower player's turn
Then cell 3 should contain 7 stones
Then cell 1 should be empty
```

The unit tests (which they called the candidate steps) are then configured to be run by the JUnit test runners (see Figure 2).

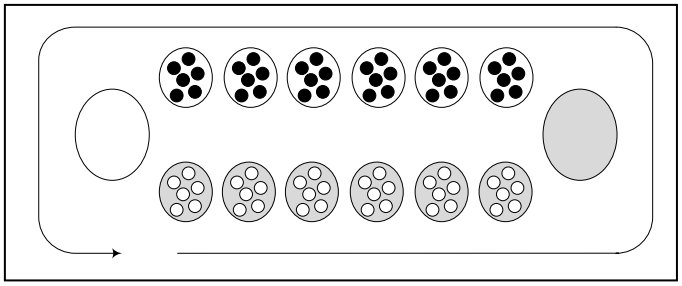

Figure 3. Kalah game: initial position

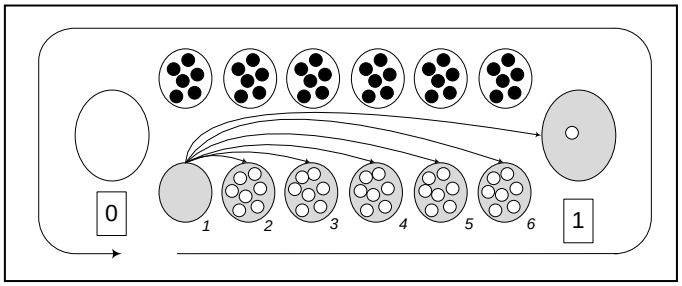

Figure 4. Kalah game: move from the cell 1

Assuming the `factory` is the game field creator, and the `field` object is used to operate with the game field, the respective unit test class methods can be implemented as follows (since the `Field` class has not been exposed, you may consider the following source as a pseudo code, although it is the fragment of compiled Java code):

```
@Given("a default game")
public void givenDefaultGame() {
  field = factory.createDefaultField();
}

@When("I take stones from cell $cell")
public void takeStonesFromCell(int cell) {
  if (field.isUpperTurn())
    field.makeTurn(cell-field.getWidth()-1);
  else field.makeTurn(cell-1);
}

@Then("it is lower player's turn")
public void lowerPlayerTurn() {
  assertFalse(field.isUpperTurn());
}

@Then("score should look like $lower:$upper")
public void scoreShouldBe(int lower, int upper) {
  assertEquals(lower,
               field.getKalahStoneNumber(false));
  assertEquals(upper,
               field.getKalahStoneNumber(true));
}

@Then("cell $cell should contain $stones stones")
@Alias("cell $cell should contain $stones stone")
public vood cellContains(int cell, int stones) {
  boolean upper = (cell > field.getWidth());
  assertEquals(stones,
               field.getStoneNumber( upper?
               cell-field.getWidth()-1:cell-1,
               upper));
}

@Then("cell $cell should be empty")
public void cellEmpty(int cell) {
  cellContains(cell, 0);
}
```

We believe that on the basis of text based scenarios the source code skeleton can be generated automatically.

```
@Given("a default game")
public void givenADefaultGame() {
  // TODO: Write the initial condition
  throw new UnsupportedOperationException();
}

@When("I take stones from cell $cell")
public void iTakeStonesFromCell(int cell) {
  // TODO: Write the unit test action
  throw new UnsupportedOperationException();
}

@Then("it is lower player's turn")
public void itIsLowerPlayersTurn() {
  // TODO: Write the assertion code
  throw new UnsupportedOperationException();
}
//...
```

Some difficulties of automated story mapping to unit tests are discussed in section IV.

III. A Study of BDD Toolkits

There are numerous toolkits supporting BDD, such as JBehave [10], NBehave [11], RSpec [12], MSpec [13],

Cucumber [14], StoryQ [15], SpecFlow [16], and CBehave [17]. Some characteristics of BDD toolkits are summarized in Table I.

TABLE I. THE BDD TOOLKITS CHARACTERISTICS

Toolkit	Analyzed Characteristics			
	Supported languages	***User stories as plain text***	***Mapping rules***[a]	***Automated mapping to the unit tests***
JBehave	Java	Yes	Yes	No
NBehave	.NET	Yes	Yes	No
RSpec	Ruby	No	No	No
MSpec	C#	No	No	No
Cucumber	Ruby, Java, Python, .NET, C++, etc.	Yes	Yes	No
StoryQ	.NET	Yes	Yes	No
SpecFlow	.NET	Yes	Yes	Yes
CBehave	C	Yes	Part.	No

a. for automated acceptance testing

As argued in [3, 9], the BDD is strongly based on the automation of the specification tasks and tests, and on proper support by the IDE toolkits. In [3], the authors emphasize ubiquitos languages, test-first practice and automated acceptance testing as key characteristics making up the BDD.

For xBehave family (represented by JBehave and NBehave), In [18] Rudolph noticed: "cycles needed to map English to executable code via attributes makes it virtually infeasible for driving out a domain at the unit level". Each sentence created in user stories has to be mapped manually to an executable method; for every change we have to find the related method and to change it respectively, and all this is really painful [20]. xSpec family toolkits have no such problems, but they are less appropriate for the business, since test scenarios are defined only in the code, not in natural langauge text or domain specific language

Even in cases where mapping rules are well defined (JBehave, Cucumber, SpecFlow, NBehave), most test systems and BDD implementations lack the support of automatic transition from plain text stories to test classes skeletons and acceptance tests maintenance.

To be fair, in SpecFlow (based on xBehave framework) there are sets of solution to facilitate writing xBehave tests and acceptance criteria, together with deeper integration with Visual Studio, including IDE templates, running and debugging facilities [17, 18].

In this paper we pay special attention to tools implementing the idea of developers/stakeholders communication framework. It is important to note that some behavior driven design instruments just implement an idea of having developer/tester communication oriented behavior modeling schema like it is in MSpec or RSpec.

TABLE II. THE BDD TOOLKITS IDE INTEGRATION

Toolkit	IDE Integration Features				
	Deploy-ment	***IDE integration***	***IDE templates***[a]	***Debug***[b]	***Unit tests***
JBehave	jar	No	No	Part.	JUnit
NBehave	Install	Plug-in for Visual Studio	No	Part.	NUnit MbUnit XUnit MSTest
RSpec	Install on Ruby	No	No	No	Ruby built in
MSpec	Install or source code	Part.	No	Yes	xUnit based
Cucumber	Install on Ruby	Part.	No	No	Ruby built in
StoryQ	dll	No	No	No	Visual Studio Unit Testing
SpecFlow	Install	Visual Studio	Yes	Yes	NUnit, Visual Studio Unit Testing
CBehave	source code	No	No	No	Own

a. like New->Fearture, etc.

b. like breakpoints on Given/When/Then and steps though acceptance test execution

IV. CHALLENGES IN BDD BASED ACCEPTANCE TESTS AUTOMATION

It seems nice to have automated procedure to pass from acceptance tests to the software unit tests. There are still some limitations and challenges that we have to consider if you try to use BDD in your team.

First, while it is easy to automate unit testing, it is far from being easy to automate conversion of acceptance tests into implementation level tests.

Second, the acceptance tests are requirements [2]. Since both requirements and software change over time, the issue of acceptance tests modification is very important. While working on a story, we may realize that some cases are being missed. Another possibility is caused with changes in the application's internal interface. The acceptance tests don't change but the conversion procedure changes.

Third, it is unclear, whether are we always able to define expected behavior that we can testwithout diving into the code inner details.

Finally, the great challenge is to progress from the unstructured natural language to the simplified structured language based on the requirement specifications and in a way that the test cases can be automated [3].

As noted above, one of the most serious drawbacks of BDD toolkits lies in their inability to convert natural language-based user stories into executable tests. Unfortunately, we have to accept this situation, since such a conversion requires

translation of informal natural language constructions into formal statements of a programming language and thus can be considered a variation of programming, which is a task for a human expert.

However, the context of BDD user stories is highly restricted, and the text of stories itself is well structured. Even the tools that are advertised as "supporting plain-text user stories" in fact set certain restrictions on the structures of stories. In particular, a user story is divided into isolated scenarios, in their turn made up of sections that represent input data, preconditions and postconditions. These structured plain-text definitions are automatically converted into executable testing code with a separate stub function for every scenario, to be implemented by a programmer.

The latter observation makes us believe that the process of user stories conversion can be automated further, at least, to some extent. While *automatic* story conversion is infeasible, individual elements of *computer-aided* conversion are certainly doable with the help of modern natural language processing methods and simple heuristic procedures.

For example, numbers, proper names, and abbreviations found in the user story most probably represent parameters to be passed to the testing code. The system can also generate sensible test function names from scenario titles.

Since people might tend to use the same words for the same objects in different scenarios, the system can analyze word use and mark (for example) most frequent nouns as potential parameters. Within this process, language processing algorithms can also recognize different word forms of the same word, which is important for natural languages with rich morphology, such as Russian.

V. Requirement Analisys for a BDD Supporting Tool

One of the important questions in regards of using BDD in design time is how to integrate better the BDD practice with the developing environments. As Table II shows, there is only limited support for BDD in existing implementations. Probably the most advanced implementation is the SpecFlow for Visual Studio, but for the case of Java based tools we found only few examples of BDD automation with quite restricted functionality. Let us take the Eclipse plug-in cited in [21] as an example. In addition to the features supported by the JBehave class framework, it allows story keywords highlighting in the behavior editor window, linking story steps to the matched unit test methods, special icon for story files, and some auto completion facilities while writing stories. These capabilities seem still not enough in regards to BDD automation.

Hence collecting the requirements for a BDD supporting tools seems to be actual problem of the domain. Here are some primary considerations for further analysis of the BDD integration solutions.

Conversion of narrative stories to the marked-up scenarios. Marked-up scenarios should use a set of predefined templates that can be converted into the source code at later stages. On the other hand, scenarios should be parts of narrative stories. It is good if narrative stories can use wide range of words and phrases from one or more natural languages.

Conversion from the marked-up scenarios to the unit tests. This task seems to be much simpler than the previous one since we need only to convert predefined templates to the code using some programming language. Usually one template corresponds to one function or method. Relation between template and method can be defined by using annotations.

Conversion from the unit tests to the marked-up scenarios. It is useful to synchronize changes made in the unit tests directly by testers or QA engineers with actual state of the user stories (it is probable that in most cases they would prefer to write code rather than stories).

"Running" user stories. To facilitate acceptance testing by the stakeholders, the unit tests may be executed as a response to the respective command applied directly to the user story.

Marking up scenarios. Creating test scenarios (and then the unit tests) may be simpler if the QA analyzer has a special automated tool to mark up the scenarios, deciding (e.g. in dialog mode) which part of user stories should be converted to which elements during marking-up stage (e.g. class and method names, method parameters, aliases, and so on).

Including meta-information to the stories. Since the BDD is about connecting developers and stakeholders, when the customer defines user stories, it may be useful to have a possibility to express relevant customer-side information. Fro example, references to the requirements specification, dependency on other stories, creation information such as data, author, reasoning, etc. may be useful.

Tracing and debugging the test executions by marked-up scenarios. Since the different marked-up scenarios may be served by the same unit test, if the scenarios are traceable and debuggable, we can easier recognize steps that failed and which data have been used in the failed tests. This feature is related to test run reporting.

Test run reporting. Test report allows us to know which narrative stories are executed correctly. In case of fail, test report should pinpoint the specific location inside the story that caused the problem. It is much easier for the tester to detect some problem if it is known where this problem occurs. It is essential to have references not only to the source code but also directly to the test scenario.

Back trace to the story from the test run. To fix a problem with some test, the developer should be able to debug an incorrectly working test. While debugging it may be useful to have associations between parts of source code and narrative stories.

VI. Conclusion

In this study we analyzed the state of the art in the domain of behavior driven development automation. Despite the fact that there are many tools supporting BDD, they are still more oriented to the developers' side, which, in some observation,

contrasts with the initial conception of facilitating communication between the development team and the customers.

We realized that in many published cases the test stories and the marked-up scenarios were composed by the same engineers (so we did, too). Therefore, those engineers may consider a necessity to write stories as some additional work that doesn't lead to unit tests that are better and easier to create.

The interest to behavior driven development is high these days. The developers are keen to simplify their job by employing software tools, and as our investigations show, there are numerous instruments to choose from. However, a closer look reveals that the most of them implement BDD ideas only at surface level. Formally they do assist agile development, but they still fail to accomplish the basic aim of BDD, namely, to simplify communications between the stakeholders and the engineers. This observation makes us believe that there exists a very perspective niche for the future development of such kind of systems. While automating the formalization of natural language constructions (i.e. natural language to formal language translation) is also the most difficult task, even modest improvements in this process can greatly increase the overall usability of BDD-supporting instruments. We advise all interested developers to have a closer look at this problem.

ACKNOWLEDGMENT

We thank Mark Finkov who encouraged us to explore the area of the behavior driven development at a period we were working on a book on software testing.

We also express our gratitude to Prof. Vitaly Klyuev from the University of Aizu for the collaboration and his valuable advices.

REFERENCES

[1] Reppert, T. Don't just break software, Make Software: How story-test-driven development is changing the way QA, customers, and developers work. Better Software, 6(6): 18–23, 2004.

[2] Melnik, G., Maurer, F. Multiple perspectives on executable acceptance test-driven development. In Proceeding of the 8th international conference on Agile processes in software engineering and extreme programming (XP'07). Springer-Verlag Berlin, Heidelberg, 2007.

[3] Solis, C., Xiaofeng Wang. A study of the characteristics of behaviour driven development. In Proceedings of 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Oulu, Aug., 30 – Sep., 2, 2011. DOI=10.1109/SEAA.2011.76.

[4] Beck, K. Aim, Fire. IEEE Software, vol. 18, no. 5, 87–89, Sept/Oct 2001.

[5] Janzen, D., Saiedian, D.H. Test-driven development: concepts, taxonomy, and future directions. Computer, vol.38, no. 9, 43–50, Sept 2005.

[6] D. Janzen and H. Saiedian. Does Test-Driven Development Really Improve Software Design Quality? IEEE Software, vol. 25, no. 2, 77–84, Mar/Apr 2008.

[7] North, D. Behavior modification: the evolution of behavior-driven development. Better Software, March 2006.

[8] XP pioneer stumps for test-first programming, In Udell, J. Test before you leap. InfoWorld, p. 55, 08.04.2003.

[9] Tavares, H.P., Guimarães Rezende, G., Mota, V., Soares Manhães, R., Atem de Carvalho, M. A tool stack for implementing Behaviour-Driven Development in Python Language, CoRR, 2010.

[10] What is JBehave, http://jbehave.org. Accessed: March 21, 2012.

[11] NBehave: BDD Framework for .Net, http://nbehave.org. Accessed: March 21, 2012.

[12] RSpec, http://rspec.info. Accessed: March 21, 2012.

[13] Getting started with MSpec, https://github.com/machine/machine.specifications#readme. Accessed: March 21, 2012.

[14] Cucumber – Making BDD Fun, http://cukes.info. Accessed: March 21, 2012.

[15] StoryQ, http://storyq.codeplex.com. Accessed: March 21, 2012.

[16] SpecFlow: Binding business requirements to .NET code, http://specflow.org. Accessed: March 21, 2012.

[17] CBehave: A Behavior Driven Development Framework for C, http://code.google.com/p/cbehave/. Accessed: March 21, 2012.

[18] What is the most mature BDD Framework for .NET?, http://stackoverflow.com/questions/307895/what-is-the-most-mature-bdd-framework-for-net. Accessed: March 21, 2012.

[19] Sanderson, S. Behavior Driven Development (BDD) with SpecFlow and ASP.NET MVC, http://blog.stevensanderson.com/2010/03/03/behavior-driven-development-bdd-with-specflow-and-aspnet-mvc/. Accessed: March 21, 2012.

[20] Borg, R., Kropp. M. Automated acceptance test refactoring. In Proceedings of WRT'11, May 22, 2011, Waikiki, Honolulu, HI, USA.

[21] Vasilev, N. Behavior Driven Development with Java, Slideshare, July, 15, 2011, http://www.slideshare.net/shadrik/bdd-with-java-8323915. Accessed: May, 22, 2012.