A Case Study on Developing a Classroom Web Application Using Behavior-Driven Development

Austin Vance and Trevor Cickovski
Eckerd College
4200 54th Avenue South
Saint Petersburg, Florida 33712 USA

Received: April 24, 2012 Accepted: May 29, 2012

ABSTRACT

Behavior-Driven Development (BDD) is a software design methodology which bridges the developer-client gap by evolving software through communication between the two sides and shaping it to the goals of shareholders. As a recently published iterative development strategy, BDD is slowly being adopted as a software practice in a wide range of domains. We study the applicability of BDD to designing Narwhal, a classroom drawing application that mimics a combination of PowerPoint slides and whiteboard. Through this case study, we employ junior and senior seminar students as clients and view the effects of BDD on Narwhal's evolution over a three-month period. We conclude with a discussion on the general applicability of BDD to the design of classroom tools following lessons learned from this case study.

I. INTRODUCTION

Mainstream software production increased with the rise of high level languages during the 1980's and at the time commonly applied the waterfall model [1], a tiered approach involving discrete stages of requirements specification, design, implementation, verification, and maintenance. This model only involved client communication in the first and last stages and although acceptable for small projects, the maintenance costs for larger frameworks that result from the client being "out of the loop" during development could be high. In fact, statistics show that traditionally 60% of software costs come from maintenance [2].

To help mitigate this problem, software developers began enforcing iterative strategies with continuous client communication and feedback between stages, encompassing a portion of the maintenance cost during the actual development. Behavior-Driven Development [3] arose from a combination of Agile Development [4] and Test Driven Development (TDD, [5]). Agile Development is quick and iterative, keeping the client in contact with the development team or a Quality Assurance Professional. Consistent contact between developer and client avoids deviating from original project goals and minimizes development lulls. Both BDD and Agile Development thus allow the client to keep software development on track and productive.

Like TDD, BDD (Behavior Driven Development) enforces the requirement of not writing any code without a corresponding test. Specifically, a prime requirement of BDD is semantic tests that have method definitions that include the word "should". This keeps tests small and specific, generally testing one requirement. By adopting a sentence structure for test method definitions, code documentation effectively writes itself and developers can clearly identify failing tests in large software frameworks. This testing environment ensures that when developers add or modify features they do not break existing features nor uncover old bugs, which in turn implies a reduced cost of programmer training on large software projects and a more fluid transition between developers of all skill levels.

We evaluate BDD's effectiveness through the development of Narwhal, a classroom drawing application that combines whiteboard and Microsoft PowerPoint. A class of nine computer science juniors and seniors played the role of shareholders, and we met once a week to

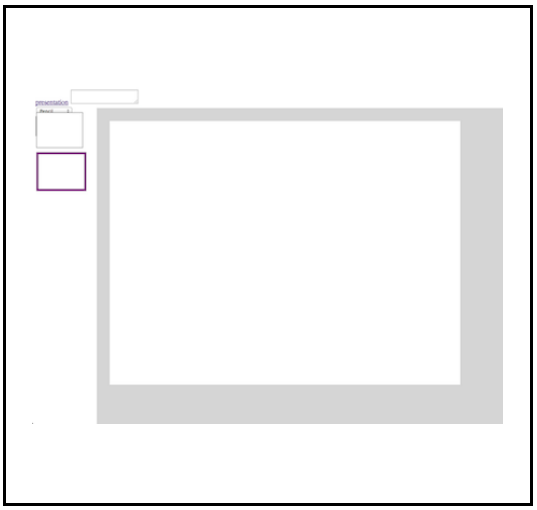

Figure 1. Early stages of Narwhal.

discuss feature specifications and bug reports. We tracked progress using PivotalTracker [6], a web application designed to monitor development of a framework that uses BDD. After receiving feature requests we developed tests using the domain-specific languages RSpec [7] and Capybara (http://jnicklas.github.com/capybara). Only after creating tests, we developed and evaluated these features in the next meeting and upon feature completion PivotalTracker notified the requestor. This cycle repeated until Narwhal met final product specifications. We share our experience with BDD as a design strategy for Narwhal, including an evaluation of its advantages and disadvantages.

II. RELATED WORK

Feature Driven Development (FDD, [8]) is an agile development methodology that orients design around individual features. A modeling team develops an overall description of the software framework perhaps using an object model and then builds a list of features that are subsequently assigned to feature teams. This is followed by a Design by Feature stage where developers refine the object mode reflect sequence diagrams developed by feature teams, followed by a Build by Feature stage where developer implement and unit test features. This strategy is similar to BDD in that it classifies features as small updates that are subsequently developed and tested, but in BDD the client determines features as opposed to the developer. Thus features become more closely aligned to the needs of the client at the cost of increased overhead for developer-client communication.

Another option is to merge Scrum [9] and Extreme Programming (XP, [10]). Scrum divides development bursts into *sprints*, with each sprint preceded by a meeting that sprint features and duration, neither of which can generally change. To keep pace with changing customer needs, Scrum is often conducted alongside XP because XP encourages daily client-developer dialogue. Most XP work is done in pairs, which in turn often compose a small group assigned to one particular feature and consistently communicating with other groups to ensure proper software evolution. TDD is also a core practice of XP [11]. While the customer communication of XP provides a lift in involvement over pure Scrum, BDD takes this further by actually involving a customer in feature construction and requiring explicit customer approval before incorporating a new feature. Scrum/XP also requires a multiple-developer environment; in this project with test BDD with a single developer.

III. THE PROCESS

Narwhal began as a small shell application with little working functionality as shown in Fig. 1. This shell provided a base for the class to brainstorm their desires for a classroom application. After conception the seminar class added features to PivotalTracker as *stories*. BDD uses ‘story’ which breaks features into separate parts as though belonging to a collection, and should have an *actor* or primary user.

Fig.2(a) shows a Narwhal story regarding canvas collaboration between faculty and student. In this case the actor is the presenter and the action is disabling or enabling the canvas. A story should also generally have a setting, or location of the action. In this particular story, the setting is the canvas. Stories should be small and descriptive, giving the client a visible outcome. Good stories are key to writing quality tests in BDD and solid client communication, while unclear stores lead to extra iterations of development and potential

(a) As a presenter I should be able to disable/enable the canvas.

(b) As a user I should not be able to edit a presentation with disabled collaboration.

Figure 2. a) Original story for allowing a presenter to block collaboration; b) Acceptance test for story 3(a).

```
let(:owner) do
  Factory(:owner)
end

let(:user) do
  Factory(:user)
End
```

Figure 3. Two mock users, an owner and a user with default permissions.

```
let(:no_collaboration) do
  Factory(:presentation, :collaboration => false, :owner => owner)
end

let(:collaborate) do
  Factory(:presentation, :collaboration => true, :owner => owner)
End
```

Figure 4. Two mock presentations, one with collaboration off and the other on.

halts. Fig. 2(b) shows a subsequent acceptance test created by us as developers, with more of a focus on technical detail. In this case we added our acceptance test as a new story, although they can also be added to the story's description depending on the level of breakdown.

Upon commencing a feature, the developer clicks a Start button in PivotalTracker which notifies the client that development has begun. The next step is to write tests for the feature, for which we used the domain-specific languages RSpec and Capybara. With RSpec, we wrote semantic tests that mimic our acceptance tests. RSpec provides a human-readable way to create tests that have clear assertions of code functionality, contributing to the client-developer communication that is central to BDD. We use Capybara to simulate user flow through a program using Selenium and Document Object Model (DOM) manipulation, simulating user activity like mouse clicks, key presses, and scrolling. By testing the part of the application that is exposed to the user we reduce the likelihood of interrupting their experience and increase the likelihood of displaying the correct error message if necessary.

To test the feature in Figure 3 we first created two mock users in Capybara, one who owns presentations (owner) and one who does not own presentations (user). We show this in Fig. 3. We then created two mock presentations, one with collaboration off and the other with collaboration on, as shown in Fig. 4.

These objects created in a **let()** isolate this test from other tests, giving each a fresh data set and avoiding cascading effects from previous tests that fail. We wrote this test prior to any application code, and then built a unit test for permissions using RSpec as shown in Fig. 5. Note the use of the word 'should', and the self-writing documentation illustrated by the clarity and conciseness of the code block.

Finally, we commenced writing application code, which initially resembled Fig. 6 prior to refactoring. We wrote a **Presentation** class with a Boolean function

```
describe "presentations" do
    it "should return true for both if collaboration is enabled" do
      no_collaboration.can_be_edited_by(owner).should be_true
      collaborate.can_be_edited_by(user).should be_true
    end
  end
```

Figure 5. Unit test for permissions with collaboration on and off.

```
class Presentation
    # ...
    def can_be_edited_by(y)
      if !self.collaboration? && self.owner == y
        x = true
      elsif self.collaboration?
        x = true
      else
        x = false
      end
      return x
    end
    # ...
  end
```

Figure 6. Application code for enabling and disabling collaboration.

can_be_edited_by that accepts a user **y** and returns true if collaboration is turned on or if collaboration is turned off but **y** is the owner, and false otherwise. Recall that this test code derived from acceptance tests, which in turn derived from stories that point to features. Each step was a checkpoint that ensured following a clear pattern from feature conception to implementation.

Although this code passes the test, we refactored it to improve clarity [12]. Refactoring generally involves breaking a method into smaller pieces to improve readability or simplify logic, renaming and removing unnecessary variables, etc. Refactoring requires rerunning tests to ensure no altering of the original functionality. We show our final application code after refactoring in Fig. 7. Refactoring is a key step because of BDD's guidelines that the developer should create the smallest amount of application code that passes the test.

We developed all Narwhal features in this manner, using this "feedback loop" between developer and client. Once we completed a feature, we clicked "Finish" in PivotalTracker which also notified the client as "Start" did, and then "Deliver" once we had migrated the code to a spot where the client could view our changes. The first three boxes of Fig. 8 illustrate this process. After delivering our changes (third row, left side), the client can Accept or Reject them (fourth row). Acceptance indicates that the feature meets all specifications and is bug free from their perspective, which leads to the lowest box and the feature migrating to production code. If the client rejects our feature, we restarted development after reading their applicable stories (third row, right side) which leads to a loop restart. PivotalTracker makes client notification easy through a Task menu, as shown in Fig. 9. This menu allows acceptance tests to be checked off, further facilitating communication.

IV. RESULTS

Throughout the semester clients became increasingly involved in the development of Narwhal, testing the application among friends and during their spare time. In the end, Narwhal grew to be

```
class Presentation
    # ...
    def can_be_edited_by(user)
      if self.collaboration? || self.owner == user
        return true
      end
      return false
    end
    # ...
end
```

Figure 7. Application code following refactoring.

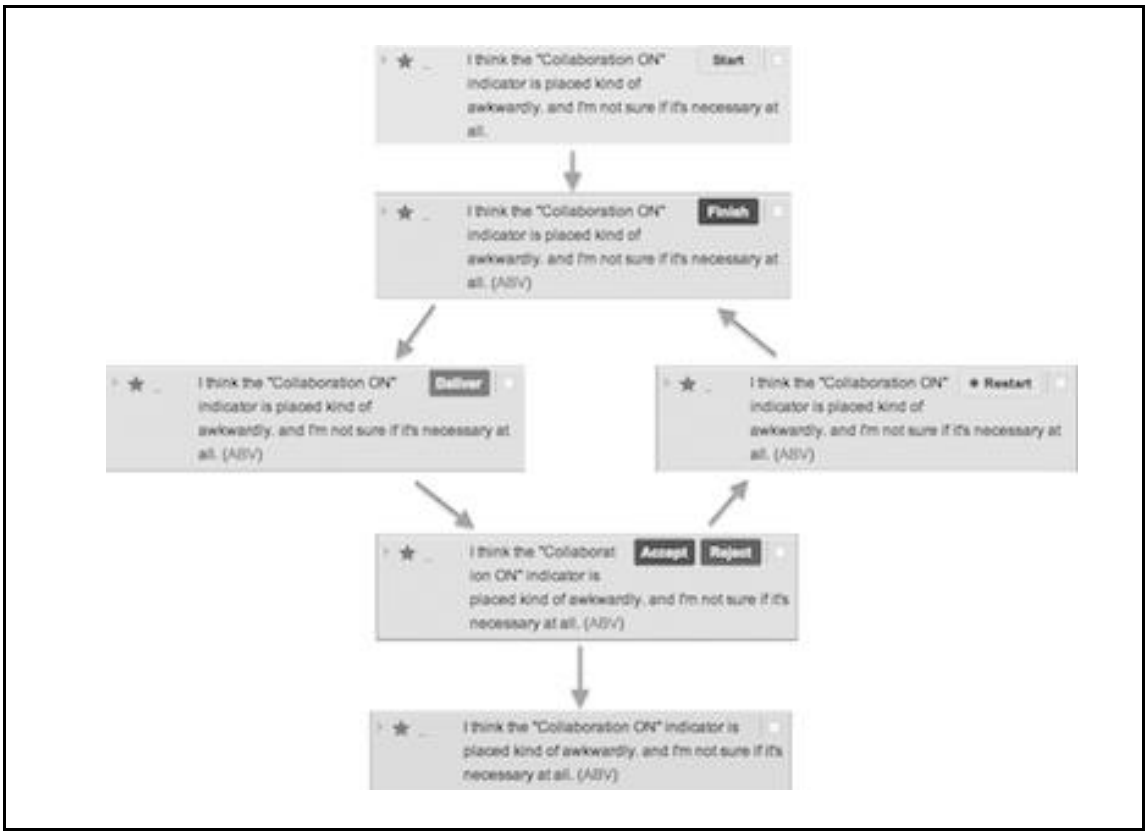

Figure 8. Feedback loop expressed in PivotalTracker.

a feature-rich web application nearly ready for deployment with a collection of features added and conceived by the seminar class, including the ability to create and modify slides in real time on the cloud (http://www.documentcloud.com). By using on-screen drawing and real-time push technology, we now have a simple plug-and-play presentation software with which anyone in the world can watch and participate, capitalizing on the power of web technology. This was completed with only one developer in three months. We show the development of Narwhal and applicable stories in Fig. 10.

A great deal of this success can certainly be attributed to BDD's consistent client communication facilitated by PivotalTracker. The feedback loop immediately notified a developer when a feature strayed from its original intent, resulting in minimal damage and no lost time. This was very evident during a story on presentation visibility, shown in Fig. 11.

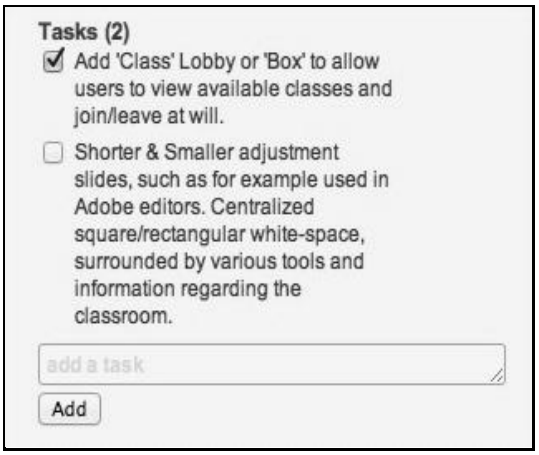

Figure 9. Task Menu applied to a Narwhal story.

The original story is a bit vague which gave us some false freedom when a lobby, which the client subsequently rejected for reasons shown in Fig. 12. After implementing the requested changes, the client eventually accepted the feature. BDD thus provided an assurance that undesirable developments that pass tests do not become highly embedded in the framework and as a result difficult to remove, saving maintenance costs. This also helped significantly during points where there was a different vision of approach between developer and client. This occurred at one point during a story about retrieving a past presentation without having to bookmark, save or remember its URL. After taking an undesirable shortcut that resulted in client rejection, the feedback loop prevented this feature from contributing to maintenance costs.

We also observed a tendency of a client to bloat their story after creation, violating the policy of stories remaining simple and clean producing the smallest noticeable changes. Many stories were rejected simply because the client wanted to add additional features. In fact, the story in Figure 3 regarding disabling and enabling the canvas was rejected later when the client wanted clear notification about the drawing canvas status.

This rejection was actually unnecessary, because the status operates completely independently of canvas enabling and disabling. There was thus sometimes a fine line between what

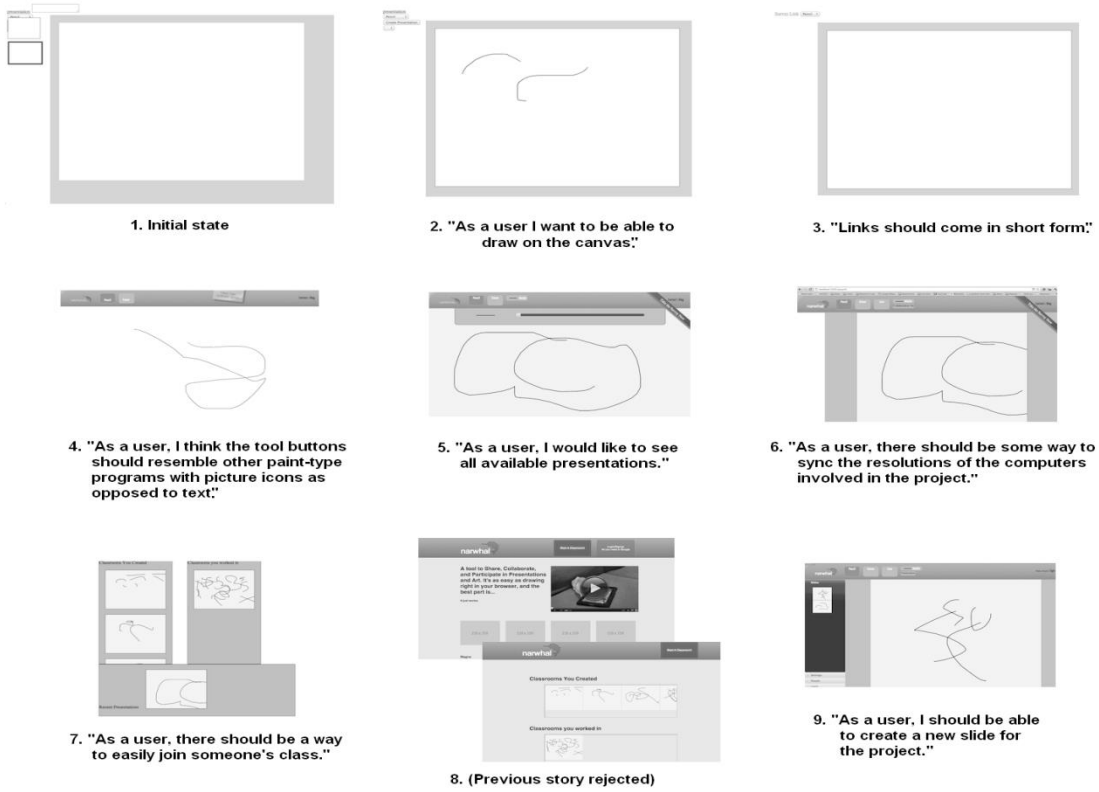

Figure 10. Key stages and stories in Narwhal's development.

(a) As a user, I'd like to see all available presentations.

(b) Perhaps in the form of a lobby? It would be nice to see the amount of people in each room, and perhaps some keywords with regards to each one.

Figure 11. (a) Presentation visibility story with (b) its supplemental description.

* Shorter & Smaller adjustment slides, such as for example used in Adobe editors. Centralized square/rectangular white-space, surrounded by various tools and information regarding the classroom.
* Faster Load times for slides
* Limit Slides based on what is expected for you to join

Figure 12. Client reasons for rejecting a visibility story.

constituted a "new" feature versus a "modified" feature. In BDD, this is generally resolved by a discussion between the developer and client on how to proceed. Eventually, we used the rule that only non-functional stories could explicitly be rejected, with story modifications creating additional stories and new features.

The weekly seminar meetings provided developer motivation especially during bug fixing. Our automated test suite written in RSpec/Capybara and PivotalTracker safeguarded against software regression. During refactoring, countless times we would incorporate a new feature and adapt to new requirements, and BDD saved us from having to remember all dependencies of the refactored code. PivotalTracker also contributed to efficient bug identification and removal through bug reports, which are derived from bug stories that work similarly to features. The developer first pinpoints the cause, breaks the problem into smaller pieces, corrects the smaller pieces, and finally automates and tests if the bug is fixed. By building tests for a particular bug into the test suite, it is much less likely to arise again and even if it should, it is highly unlikely to slip through uncaught.

V. CONCLUSIONS

Narwhal demonstrates that BDD can be used to develop a sturdy piece of software, control development tangents, keep the developer on task, and help developers tackle complex problems by breaking them into smaller pieces. Following BDD's procedures, we completed each step of Narwhal in a generally efficient fashion.

BDD did have some limitations with a single developer and because the seminar class did not have time for additional meetings, weeks would sometimes pass without significant progress if a feature happened to be time-consuming. However, considering that a large percentage of software projects die before reaching deployment [13] we believe the situation would have been worse without the feedback loop because the lone developer would have been easily sidetracked by unnecessary or undesirable features. Indeed upon completion of the seminar, several students remarked on the attractiveness of BDD simply because they felt their opinion was highly valued and that they could witness more closely application progress. Many also remarked on the efficient application development resulting from the feedback loop. Several others who had previously worked on software internships where they had been expected to learn a large amount of code in a short period of time expressed the merits of human-readable semantic tests.

In addition, we also learned that BDD's success is extremely dependent on active client and developer dedication. If either party is negligent in communication, this breaks the feedback loop and can halt development and progress. Since the seminar class was coerced with bonus points but had nothing to gain by Narwhal's success, their interest in the development process was sometimes limited resulting in an ultimate lacking of original perspective and necessary priming on features and

stories. Fortunately development never halted but some features strayed a little farther from the original goal than probably necessary, eventually solved by the feedback loop. This problem became less significant throughout the semester after educating the seminar students about the vitality of their input and its implications. Indeed, as suggested by Liz Keogh [14], a key proponent of BDD is that clients understand the *business value* of a software framework. This will motivate them to make sure when developing and accepting or rejecting features that this value is being properly delivered. Even North himself further suggests that an important question to consistently ask clients is "what is the next most important feature that the software does not do?" This allows for feature prioritization in terms of business value and provides clients with a starting point even if they will not directly benefit from the software's success.

Although BDD's self-correcting nature allows companies to produce more advanced software at lower costs, it can initially be a hard sell for clients because of the burden required throughout the development. It can be challenging for clients to remember that there is also a cost of evaluating the end product, and as developers we believe the overall cost is less when considering this extra amount. Although the extra work involved in tests and communication may seem to divert from implementation, the clear steps outlined by BDD keeps developers from getting stuck. We often found ourselves asking the "5 whys" [15], for example:

I need a login form. (Why?)
So a user can identify themselves. (Why?)
So a user can access profiles they created, and keep them private. (Why?)
So a user's work is only editable by them.

Through this feature deconstruction we could find a clear starting point and identify parts of a feature. This is particularly useful for new developers, helping them to divide and conquer a large problem and attain visible success. This also ensures that the most fundamental parts are built first as a foundation followed by higher level features, and encouraging modularity and flexibility. Later savings from code engagement and bug identification which create a comfortable environment that ensures modifications do not break existing features also outweighs the overhead of new developers learning the automated test suite. This lessens the likelihood of the loss of "experts" on a large software project halting development until someone else can master the application code. Teams can develop features inside a "box", modifying the project code upon feature deployment which only occurs after all tests pass.

Narwhal identified reasons why BDD should be considered as a viable software development strategy. Tools like PivotalTracker minimize extraneous effort and help drive development forward through the feedback loop between client and developer. Languages like Rspec and Capybara further bridge this gap, focusing on natural language and concise syntax. A future goal would be to repeat this process but involve people truly invested financially in project success. This would more clearly identify the advantages of BDD and also enable comparison with competing software methodologies like TDD, the results of which could help clients and developers to choose the strategy that best suits their needs.

REFERENCES

1. W. Royce. Managing the development of large software systems. In *IEEE WESCON,* volume 26, pages 328-338. TRW, 1970.
2. R. D. Banker, S. M. Datar, C. F. Kemerer and D. Zweig. Software complexity and maintenance costs. Communications of the ACM 36(11):81-94, 1993.
3. D. North. Introducing BDD. In *Better Software Magazine*, March 2006.
4. L. Cao, B. Ramesh and T. Abdel-Hamid. Modeling dynamics in agile software development. ACM Trans. Manage. Inf. Sys. 1(1):5.1-5.26, 2010.
5. K. Beck. *Test-Driven Development By Example.* Addison-Wesley, 2003.
6. PivotalTracker: Simple, Agile Project Management Software and Team Collaboration.
   See www.pivotaltracker.com.
7. D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp and D. North.

The RSpec Book: Behavior-Driven Development with RSpec, Cucumber and Friends. Pragmatic Bookshelf, 2010.
8. S. R. Palmer and J. M. Felsing. A Practical Guide to Feature-Driven Development. Prentice Hall, 2002.
9. K. Schwaber and M. Beedle. Agile Software Development with Scrum. Prentice Hall, 2002.
10. K. Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.
11. What Is Extreme Programming? See http://xprogramming.com.
12. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
13. R. Charette. Why Software Fails. *IEEE Spectrum* 42(9):42-49, 2005.
14. L. Keogh. BDD/TDD Done Well. See http://lizkeogh.com/2007/06/13/bdd-tdd-done-well.
15. K. Bulsuk. An Introduction to 5-Why. *Karn Bulsuk: Full Speed Ahead*. Published online March 2009. See http://www.bulsuk.com/2009/03/5-why-finding-root-causes.html