Narwhal:
A Case Study Developing a Web Application for the Classroom
Utilizing Behavior Driven Development

by
Austin Vance

A Senior Thesis in Computer Science
Submitted in Partial fulfillment of the requirements
for the degree of
Bachelors of Science

at

Eckerd College
St. Petersburg, FL

January 23, 2012
Committee Report

A Thesis in Computer Science

This Thesis was prepared under the direction of the Research Advisor and the candidate's Thesis Committee. It was submitted to the Faculty of Eckerd College and was approved as a partial fulfillment of the requirements for the degree of Bachelor of Science.

January 23, 2012

THESIS COMMITTEE:

______________________________
Research Advisor (Dr. Trevor Cickovski)

______________________________
Dr. Kelly Debure

______________________________
Dr. Nazarré Merchant
Acknowledgments

Thank you without everyone below this Thesis would have never gotten completed:

Dr. Trevor Cickovski
Dr. Richard Adams
Andi Berry
Desi McAdam
Lance Eppley
2011 Fall Seminar Class
Sheehan Commette
Les Hill
Jon Larkowski
Veezus Krist
Corey Haines
MityMo
...
and Especially My Dog
Boz
Abstract

In this thesis I present a case study of the value of utilizing Behavior Driven Development (BDD) to develop a classroom presentation tool, Narwhal. Using a junior and senior seminar class in Computer Science, Narwhal was developed using Acceptance Testing, Unit Testing, and Integration Testing. The students were responsible for creating and guiding the application’s development through weekly meetings and feature planning using Pivotal Tracker. After the semester of development Narwhal grew into a fully functional drawing application that utilized HTML5 technology such as the Canvas API and WebSockets. Through this experience I evaluated BDD and its applicability in client driven software projects. After the research was completed I came to the conclusion that BDD demonstrates a powerful ability to help build software projects by keeping the developer on task, increasing client and developer communication, and creating clearly defined features and tests.
# Table of Contents

Chapter 1: Introduction ........................................................................................................7
  s.1.1: Introduction to BDD ..........................................................................................7
  s.1.2: Why use BDD ..............................................................................................8
  s.1.3: Approach .......................................................................................................10

Chapter 2: Background/Related Works ...........................................................................11
  s.2.1: What is BDD ..................................................................................................11
  s.2.2: History of development methodologies .......................................................14

Chapter 3: Narwhal .......................................................................................................17
  s.3.1: The Current Void ..........................................................................................17
  s.3.2: Implementation .............................................................................................19
    s.3.2.1: Feature Discussion .................................................................................21
    s.3.2.2: The Story ................................................................................................24
    s.3.2.3: Testing ......................................................................................................27
      s.3.2.3.1: Acceptance Tests ..........................................................................27
      s.3.2.3.2: Integration and Unit Testing ..........................................................28
  s.3.3: Application Code ............................................................................................32

Chapter 4: Results .........................................................................................................38
  s.4.1: The Feedback Loop ........................................................................................38
  s.4.2: Student Stories and Their Effects ..................................................................43
    s.4.2.1: Breaking Stories Down .........................................................................43
    s.4.2.2: Immediate gratification .........................................................................46
    s.4.2.3: Development from Start to Finish ........................................................48
  s.4.3 Shortcomings of Driving by Communication ..................................................55

Chapter 5: Conclusions .................................................................................................57
  s.5.1: The Creation of Narwhal ..............................................................................57
  s.5.2: Importance to Computer Science and Software Engineering ....................59
  s.5.3: Utility to Large Software Projects and Open source ....................................61
    s.5.3.1: Improvements in the BDD process ......................................................62
  s.5.4: Future Work ..................................................................................................63

Bibliography ................................................................................................................65
Chapter 1: Introduction

s.1.1: Introduction to BDD

To develop software, companies have tried many different techniques. In the past, programming an application was thought of as a solving a problem, and the method for solving the problem was thought to only be as good as the software that was created. As higher level languages evolved, software development became more popular and software became increasingly mainstream. Therefore the method that a developers used to create software came into question. Behavior Driven Development (BDD) was born out of that evolution. BDD has a strong focus on collaboration during software development, much more than it’s predecessor Test Driven Development (TDD).

BDD's focus does not end with developer collaboration, but instead attempts to bridge the gap between the developer and client, where features of an application are defined by how a user would interact with them rather than a feature being defined in technical terms. BDD bridges the gap between the developers and the stakeholders in software by providing a clear concise methodology to develop software.

The basic work flow for software developed using BDD techniques requires the developer to communicate with the shareholders, and it requires developers to create a well defined picture of the application. Then, through the development process, shareholders communicate with the developer to shape each individual
feature to the specific goals of the shareholders. This approach creates a dialogue between the developer and shareholder to create an end application that meets the specifications much more closely than applications developed in any other way.

s.1.2: Why use BDD

Developing software using BDD has several advantages. The most obvious advantage is when developers start to add or modify a feature, they can be sure their modifications will not break any other features and uncover any old bugs. For example in a To-do application that tracks a list of tasks a user enters, after the developer implements the feature for adding To-do's to a list, a client now wants to add the ability to add a due date to each To-do. If the developer is not using BDD, when the developer modifies the code that adds a new feature to take a date as well, and they could create a bug in the ability to add a To-do. It is that developer's job to make sure that every feature works. On the other hand, if using BDD, the developer would just have to run the old tests that were designed for the prioritization feature and then immediately realize there was a problem. On a small application this may unnecessary, but on an application consisting of hundreds of thousands of lines of code and thousands of features, making sure everything works after every change is impossible without BDD.

Another clear advantage of using BDD similar to preventing regression in features, and this is the ability to bring new people into a project and get them work-
ing on features without risk of creating bugs in the larger application. For instance, when a company with large applications hires new software developers there is a huge cost that comes from bringing the new developer up to speed. Companies that do not employ BDD require that a new developer learn the entire application before developing code. This is extremely time consuming. By using BDD, a new developer can quickly learn the necessary features, that help them achieve their goal. They begin to implement features and run the programmatic tests that are necessary to test new features. If the tests show a bug in the larger application, then the new developer can modify the new code accordingly. Testing allows for a much more fluid transition between developers, and even allows for open source collaboration between developers of all skill levels. BDD excels in that the new and less skilled need not develop a fear of breaking the more advanced features.

One final advantage of BDD is that it allows any developer to refactor without fear of breaking any feature. Refactoring is a key step in software development; it allows changing code to be more readable, maintainable, and modifiable. When refactoring code that is not part of an application developed using BDD the feature can have no clear boundaries, and modification could break any larger feature that depends on it ("The Rspec Book" 54). For example when developing a calculator, a developer might want to refactor the addition feature so it would not display any trailing zeros after a decimal place. If another feature, such as division requires all numbers to have two decimal places, the refactor would break
that feature even though the overall functionality was not changed. BDD would allow for a developer to run the test suite for the whole application after refactoring a feature to make sure every other feature still works in harmony.

s.1.3: Approach
To evaluate the effectiveness of BDD, I developed Narwhal, a drawing application designed to mimic the combination of a whiteboard and a PowerPoint Presentation. During the development process, a class of nine computer science juniors and seniors were polled once a week to create new feature specs and bug reports. In the development of Narwhal, the students in seminar played the role of the shareholders during the development phase. The students created stories in Pivotal Tracker, a web application designed for tracking the development of a piece of software that is employing BDD. After putting stories into Pivotal Tracker, tests for the software were created using the software suites Capybara, Rspec, and Jasmine to test different parts of the software respectively. After tests were created for a feature, the feature was developed and evaluated in a client/developer meeting. From the testing session more stories were added until the specifications for the final product were met.
Chapter 2: Background/Related Works

s.2.1: What is BDD

BDD is a software development methodology that stems from a combination of Agile development and Test Driven Development. It was originally conceived by Dan North in the November 2006 issue of Better Software. North spends his article discussing how a focus on behavior can aid the production of software more than competing philosophies. First, North discusses BDD’s requirement for semantic tests. BDD requires that tests have method definitions that include the word “should.” The usage of the word “should” keeps tests small and specific because “should” forces the use of only one requirement.

North goes on to discuss that by having sentence like-method definitions in tests, the documentation for code is already writing itself. This holds true in the Rspec shown in Code Block 2.1.
describe "In the person class" do
  it "the name method should return a combination for the first and last name" do
    # ...
  end
  context "the age method" do
    it "should return a number" do
      # ...
    end
    it "should not be negative" do
      # ...
    end
  end
end

The documentation for the code could easily be created with a parser that takes the definitions of the describe, context, and it blocks. Documentation for Code Block 2.1 that could be auto generated can be found in Figure 2.2.

**Figure 2.2**

<table>
<thead>
<tr>
<th>In the Person Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>The name method should return a combination for the first and last name</td>
</tr>
<tr>
<td>The age method</td>
</tr>
<tr>
<td>Should return a number</td>
</tr>
<tr>
<td>Should not be negative</td>
</tr>
</tbody>
</table>

With this clean and pre-created documentation, code is maintainable and concise.
North goes on to describe the utility of these simple tests. Sentence like tests allow developers to identify very clearly what test failed and where it is located. This coupled with meticulous file organization makes finding a failing test even in large code suites a simple task.

BDD's wording also helps ease a developer's worry about modifying the test suite. If the test definitions are not clear, new developers are tentative about deleting tests even when they are not necessary anymore. For example:

**Figure 2.3 ("Introducing BDD")**

- I had introduced a bug. Bad me. Solution: Fix the bug.
- The intended behavior was still relevant but had moved elsewhere. Solution: Move the test and maybe change it.
- The behavior was no longer correct – the premise of the system had changed. Solution: Delete the test.

North explains that the first of these is a common practice of development, but the others are normally not completed by new developers. The inclusion of the word "should" clearly defines the test's purpose and can help developers ease their mind when modifying the test suite and the code.

BDD also focuses heavily on the necessity of client communication. This set of requirements is inherited from its parent methodology, Agile development. Agile enforces quick iterative development (Cao et al. 4-5). These iterations help keep the client in contact with the development team or a Quality Assurance Profes-
sional. Consistent contact between client and developer helps keep the overall project in control of the client and avoid forgetting or misinterpreting original project goals. Lulls in development can also occur but by repeatedly reporting to a client features are developed faster and in a clean, presentable manner. Agile development and BDD place the burden on the client to keep development on track and productive.

BDD is a combination of Test Driven development (TDD) and Agile Development. With TDD no code is written without having a corresponding test first. BDD takes this and expands it to use North's procedures for writing tests. When combined with Agile's development iterations and client communication, BDD is a very powerful and effective methodology for writing software.

s.2.2: History of development methodologies
In "Introducing BDD" North describes the downfalls of older development methodologies and how BDD attempts to solve them. Agile Development is a term used to define a modular approach to development. These incremental approaches have been used since the late 50's. Agile Development even has a website where developers can sign a manifesto.

Figure 2.4 (Beck et al.)
- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan
This creed is signed by over 5000 developers that are dedicated to creating software in an efficient way that is not only functional but also maintainable. The Agile philosophy focuses on communication with the client and clean, iterative development. Agile development also descended into development strategies known as Extreme Programming (Che 83). Extreme Programming and Agile Development use iterations for development just as Narwhal iterated thorough a full development cycle in one week, from conception to production. Before the iterative development strategy that Agile Development utilizes, development was normally called a waterfall model (Smith).

**Figure 2.5** (Smith)

The waterfall model was a tiered approach that involved little client communication. The Waterfall Model only involves the client during the “Requirements” and
“Maintenance” stages, while Agile Development involves the developer during every step of development. After the project is defined, the design process begins and software is written. After the project is complete it is sent to the client for verification and finally acceptance. After acceptance the code enters maintenance mode, where it is only modified to fix bugs that arise and maintain functionality. This methodology is acceptable for small projects but lacks the maintainability of TDD and BDD, as well as the iterative approach and constant client communication encouraged by Agile. BDD has a strong focus on keeping maintenance as part of the entire development phase, rather than a step at the end, this is an attempt to mitigate the fact that 60% of the cost of software comes from maintenance (Banker et al. 81).
Chapter 3: Narwhal
s.3.1: The Current Void

With current advances in web-based technology, and real-time collaboration without the need for third party "out of browsers" application programming interface's (API) Pandora’s box has been opened for web applications. Companies have been focusing their effort on fully browser integrated document editing and spread-sheet tools. Document Cloud (Document Cloud) has created a real-time collaborative text editor. These real-time applications take advantage of new the new HTML5 API (HTML), using sockets to push information from a server to the browser.

Before the creation and standardization of WebSockets, a technology to allow servers to send information back down to a browser without a waiting, open request (HTML5), Web applications that took advantage of push technology had to use third party API's and browser plug-ins like Flash and Java applets. The reliance on third party technology causes browser lag and requires a user to install extra software on their computer just to use an application. One popular piece of software that Eckerd College utilizes is called Ubiquitous Presenter (UP). UP uses Java applets to create a drawing surface as well as long polling for real time communication. These two implementations cause a very simple application to flood any network with requests when more than a few users are trying to participate. The solutions that UP and other companies are offering for real-time applications are usable, but have a very broad client scope and are closed source,
or are not the correct technology for the project. The potential for creative applications that could aid education seemed like an obvious step in the evolution of the work discussed in this thesis. With the need came a project called Narwhal.

Narwhal is a web application developed in unison with the research for this thesis. There was a clear need for a reliable web application that focused on integrating the roles of student, teacher, and technology.

Currently the most used presentation software is Microsoft’s PowerPoint. This was an excellent piece of software when it was released and still is a world leader for enterprise presentation software, but it does not meet the specific needs of a classroom. Instead education should be thought of as a communication between student and teacher. When the education system uses a discussion about a subject rather than a purely presentation driven lecture, students absorb more material. PowerPoint does not provide the any sort of back-and-forth between student and teacher (Jones).

To meet this need, Narwal was developed. Narwhal was structured to be a collaborative presentation tool designed using Behavior Driven Development, with students and professors as the stake-holders. Each week development [iterations] (Agarwal 52), short periods of development that produce noticeable features, were presented to the seminar class. The class evaluated and determined the effectiveness of each new feature, and then proposed new features for the next
sprint. After receiving new feature requests, the features were entered into a tool called Pivotal Tracker (Pivotal) and set up as acceptance tests (Martin). Pivotal Tracker is a web-based technology that is designed to provide feedback to the client or feature creator throughout the development process. Each acceptance test was linked to a specific requester in the focus group and as the feature was completed, the requester was notified and asked to evaluate the feature. Features, a noticeable addition to the overall project, follow a strict process, using very classical BDD methodology.

s.3.2: Implementation

After a feature’s conception, the acceptance test is discussed and written, and the developer begins by writing a programmatic test in a Domain Specific Language (Robles Luna 297). The test would describe and check the functionality of the features in the top level of the user experience. After writing the outermost tests, more specific tests are written as needed in a different DSL designed to test algorithms rather than user experience.

The first time Narwhal was presented to the focus group, it was a small shell application that had very little working functionality.
Figure 3.1 illustrates a stage of development where there was no functionality including drawing, it was just a sample of a possible layout.

Basically, it was a proof of concept, similar to a piece of software that would be shown to a client during a bidding process. The shell of the application was used to increase the seminar class’s interest and willingness to brainstorm about features they would desire in an application that would ultimately be used in the classroom. In time, individuals in the class became more involved in the development process, and even began testing the application among friends and using
it in their spare time. It is this stake in the software development process that provides a strong case for BDD.

The combination of on screen drawing and real-time push technology allows for a very simple plug-and-play presentation software. As a presenter draws on screen highlighting and amplifying a pre-created presentation, or creating a presentation on the fly with a pen tool, anyone, anywhere in the world can watch and participate in real-time. This is the true power of web technology.

s.3.2.1: Feature Discussion
Narwhal changed drastically after its first presentation to the focus group. The shell of an application grew to become a feature-rich web application nearly ready for deployment. In the end, Narwhal, a powerful web application, has a large collection of features all added and conceived by people originally invested in the product, the stakeholders. Narwhal, in the last stages of working with the client, is a full blown presentation application, with the ability to create and modify slides in real-time in the cloud. It takes advantage of HTML5 technologies including the canvas element to have pixel based drawing and performance. These technologies provide a cutting edge efficient approach to real time drawing in the browser.

The HTML5 canvas is the core to the user experience in Narwhal, and provides a rich API for drawing and creating lines in a browser window. It also exposes the
core of its API and pixel data which was imperative for using BDD when creating Narwhal. All other existing drawing technologies for the web such as SVG, Flash, and Java, only expose through their own software, or in the case of SVG no pixel data is exposed, and instead vector based images are created. When drawing a line on the screen and tracking mouse movement, the pixel graphics are much lighter for the browser and allow for faster mouse tracking. Research from University of Helsinki have created proof of concept applications that allow a user to draw graphics using SVG, but they suffer from performance drops as every square added to the screen is a new vector graphic, and only after a mouseUp event is fired will the graphic drawn be post processed into a single vector (Kaipiainen).

The same effect is achievable with the canvas using the lineTo API to darken pixels along the mouse path. A simple example of code to draw a line on a 100 x 100 canvas would look like Code Block 3.1.

```
canvas = getElementById('canvas')
ctx = canvas.getContext('2d')
ctx.moveTo(10,10)
ctx.lineTo(90,90)
ctx.stroke()
```
This code is simple, clean, and expressive; the first two lines are necessary to instantiate the canvas API and create the proper listeners on the canvas for drawing. After initiation, the `lineTo` method provides easy and explanatory code.

The use of the canvas element did include some flaws, the major one being that it is extremely new technology and not 100% cross browser compatible. No Internet Explorer browser earlier than version 9 can see the canvas element. This problem was discovered and handled by the seminar class. When the group was testing the new drawing functionality, some of them used Internet Explorer and said the feature was completely useless. After examination I was able to find the problem and work out solutions based on client’s needs.

Along with drawing on the screen, Narwhal provides the ability to share in real-time what a user writes and draws on the canvas. To do this Narwhal uses a third party service called Pusher (Pusher). Pusher is a service that developed a Ruby-Gem and a Javascript library for working with WebSockets and falling back to Flash, when HTML5 is not supported. Pusher is the key to cross-browser compatibility of the feature; the use of third party technology allowed for faster development of the feature and reliability. It also has trade-offs that had to be explained to the seminar class. Third party software includes the lack of ability to expand and fix bugs, and requires reliance on someone else’s service. This heavy reliance forces the application to be at the whim of third party outages and failures as well as our own.
Discussion of these trade-offs is a key part of the client communication in BDD. When implementation details can be explained to the client, it is imperative that the developers explain why they chose the route they did and also provide explanation of other possibilities. Given the developer’s explanation, it becomes the client’s job to determine if the current implementation is acceptable or if a new implementation is necessary. In the case of using third party libraries, it is the developers job to explain the consequences:

**Figure 3.2**

| - Loss of control  
| - Speed of implementation  
| - Reliability of service  
| - Degree of integration |

It is also important to consider how the development and integration of the 3rd party software can be tested. Will it be as reliable as the rest of the application? Some of these decisions are clear but must be considered every time a library is used.

**s.3.2.2: The Story**

After working through conception, each feature created by the focus group was added to an Agile feature tracking software called Pivotal Tracker. After a feature is conceived, it is added to Pivotal Tracker as a *story*, the word story is used in BDD to describe the desired feature because each feature should have separate
parts, just like a story in a collection. Each feature has an actor. The actor is the primary user for the feature. If the feature is a user administrative panel, the actor may be a manager or a website administrator. If the feature is user log-in, the actor would be any person not logged in. After a story has an actor there must be an action the actor would take. Features should be small steps that create a much bigger picture. Actions for the above examples could be an administrator wishing to create a user, or a user wishing to log in. These two examples take the first part form of a feature's story. Along with an actor and action, a story must have a setting. The setting is a description of where the action should take place. An administrator should be able to create a user on the user admin page; or a user should be able to log-in from the home page. With these three parts you have the most basic skeleton of a feature's story, also known as an acceptance test. Good stories are the key to writing good tests for your BDD, and also key to client communication. Unclear stories lead to extra iterations of development and halted processes. Stories have two main requirements, they should be small, and they should be descriptive. Each story should also give the client some sort of visible outcome. An example of good stories are in Figure 3.3.

**Figure 3.3**

- As a user I should be able to log-in from the home page.
- As an administrator I would like to be able to edit user information from the user admin panel.

Both of those stories have all the required layers and key descriptors to implement the feature, and are all the story needs.
After working with the client and entering a story into Pivotal Tracker, a developer must break the story down into acceptance tests. A story, although small, may require many steps to complete the entire feature, each step should be added in the same fashion as the above story, but with more focus on technical detail. This step is not for the client, or to improve client communication, but instead to help a developer discover all the necessary implementation details in a feature. For the story describing a user logging into a website the stories would break down to:

Figure 3.4

- As a user I would like to be able to log-in from the home page
- The home page should have a log-in form with two input fields, User-name and Password.
- Submission of input on the home page should check the User-name and password against user table
- If the user is verified they should be redirected to their profile page
- If the user name and password are incorrect the user should be returned to the home page with an error message.

Each of the above steps smaller break downs of the full feature story. In Pivotal Tracker they are either added as part of the description or as new stories depending on the size of the break down. Once a story cannot be broken down any further it is rated.

Agile development tools like Pivotal Tracker have developers rate stories based on difficulty ranging from zero to eight. Zero points is extremely easy, something like changing the word in a Submit button may be a zero point story. Eight point stories are considered too large and normally contain many smaller stories that
should be broken down. The point scale is a reflection of the expected difficulty of a story after the original story has been analyzed. Point rating should not reflect the amount of time required to complete a story, as it is extremely hard to predict and stories should always take a minimal amount of time if broken up correctly.

s.3.2.3: Testing

s.3.2.3.1: Acceptance Tests
After point estimation on Pivotal Tracker, a developer can actually begin to write code. Before beginning on code, the developer clicks Start, which will indicate to the client that the development process for this story has begun.

After clicking Start the developer will write a test describing each of the stories in the feature. For the user log-in story there are two layers to testing. First is top level testing, or behavior testing. Behavior testing separates BDD from its predecessor TDD. The behavior tests should reflect user actions such as filling in forms and clicking on buttons, and should take user paths. For Narwhal, RSpec (RSpec) and Capybara (Capybara) were used to create the top level behavior tests. RSpec is an open source testing library designed to have syntactic tests describing user action. They describe themselves as a tool "to [aid] software development that combines Test-Driven Development, Domain Driven Design, and Acceptance Test-Driven Planning" (RSpec). The key feature that RSpec provided for Narwhal is the ability to write semantic tests that mimic acceptance testing. By creating
tests that have more clear assertions of code function, and a domain specific language that is human readable, the gap between client and developer is closed significantly. It is a key feature of BDD that the client and the developer think on the same terms.

s.3.2.3.2: Integration and Unit Testing
After acceptance tests were entered into Pivotal Tracker, the Narwhal developers would write semantic tests using Capybara to top level test any parts of the story on Pivotal Tracker that will be exposed to the end user. Capybara allows a developer to automate user flow through a program using Selenium and DOM manipulation (Capybara). The advantage to using Capybara over just Rspec to test an application comes from the behavior testing part of BDD. The key component of BDD is the fact that user behavior is tested, not just the core of any application. With Capybara and the ability to mimic user actions such as clicks, mouse moves, key presses, and scrolling becomes a part of the already established Rspec library. Advantages are more clearly outlined when actual test code is written and run. With Rspec to test the user login page, an object method must be created and it either returns a result that Rspec can monitor, or throws an exception that Rspec catches. Both of these options are impractical when testing something simple, for example when a password field is de-focused an error should be displayed if the entered text is too short. To test an event like this with Rspec but not Capybara the developer would be required to create a method that would be accessible to the test suite and return a value that is testable. In a Ruby
On Rails model, this would require a very involved length checking mechanism, with AJAX responding to the server to trigger events and unnecessary HTTP requests. The code required at minimum would look something like displayed in Code Block 3.5.

```ruby
def good_password?(password)
  password.length > 7
end
```

This code looks extremely simple but it lacks any user interaction and requires extra Rails tests to make sure the AJAX request

```ruby
$.get('/good_password', { password: ?? }) // With ?? being any entered password.
```

followed the proper application path, accessed the correct model, and returned a correct value.

This is not a feasible solution, before TDD instead just the method is checked using unit testing as shown in Figure 3.7
To test this method with Rspec it is as simple as

```
describe "the password" do
  it "should have a length greater than 7" do
    user = User.new
    user.password = "hello"
    user.good_password.should be_false
    user.password = "over_seven"
    user.good_password.should be_true
  end
end
```

The test above is comprehensive because it covers all the outcomes of a length check on a password, but it lacks any sort of real application testing. Just because the developer tests that a good_password method returns correctly, the developer cannot be sure that the good_password method is fired when the user un-focuses the password box.
Instead, when using Capybara, which can evaluate the Document Object Model (DOM), the tree of HTML elements, and run any client interaction, the test is simple as shown in Figure 3.9.

**Code Block 3.9**

```ruby
describe "the password blank" do
  it "should display an error when the length is less than 7" do
    fill_in "Password", :with => 'short'
    page.should have_content("Password is too short")
    fill_in "Password", :with => "over_seven"
    page.should_not have_content("Password is too short")
  end
end
```

The key difference between the tests using Rspec alone and tests using Capybara with Rspec is the latter testing actually inspects the page that is generated by code. To a user, the functionality of a method buried deep into the back end of an application is not nearly as important as the correct error actually being displayed.

Another clear advantage to testing the part of the application that is exposed to the user is the assurance that the user experience is never interrupted and never fails. Narwhal has several back end scripts that run to keep the database efficient, as well as remove unwanted data. These scripts are not essential to the functionality of the application. When executing a user feature to add the ability
to create new slides, tests for the back end scripts failed, as well as a user level test that checked to make sure lines drawn were synching. I decided to fix the back end scripts and tests before I presented to the focus group, rather than the front end. This decision had the consequence of stories on Pivotal Tracker being rejected by the focus group as well as a new wave of bug stories being entered.

s.3.3: Application Code
During the entire process of writing tests, collecting stories and getting feedback from users, application code has to be written. BDD has very strict guidelines about how application code is written after tests have been created. The first goal of a developer is to create the smallest amount of application code possible to pass the test. This first step is very counter-intuitive for developers who do not utilize BDD regularly. BDD focuses on the smallest parts of problems breaking each down into a unique test. The easiest way to demonstrate the correct way to tackle a problem is to proceed through the development process of a feature in Narwhal. In the beginning of the Narwhal development process, a feature was requested to allow a presenter to block collaboration on the canvas, allowing only the creator to draw on the presentation. The feature was added as a story saying...
Figure 3.10

- As a presenter I should be able to disable/enable the canvas

This was the entire description given by the focus group. It follows the basic idea of a story and is very clear about the desired goal. But the story is not concise. Instead it was broken into sections. The section of the story to focus on is:

Figure 3.11

- Presentations with disabled collaboration can only be edited by their creator.

The test is then written by first creating two mock Presentations, one with collaboration disabled and one with collaboration enabled, two users are also mocked: one who owns both presentations and one who does not own either.

Code Block 3.12

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

let(:user) do
  Factory(:user)
end

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

let(:collaborate) do
  Factory(:presentation, :collaboration => true, :owner => owner)
end
```
These two `let` calls create a mock using the `Presentation` class that have default values assigned. Using the mocks allow tests to be isolated from other tests, this way each test gets a fresh set of data to work with, and isn't seeded with bad data from a failing test before it. Then tests are written before any application code is written.

```ruby
Code Block 3.13

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
```

The important thing to note is that these tests assume the `presentation` has the method `can_be_edited_by`, and that the method takes a user as a parameter. Even though those assumptions are made, the `Presentation` class may not even be written and the `can_be_edited_by` method has almost assuredly not been created.

The concept is that the tests are clean, concise, expressive, and model the code that the application will need to have. When following this pattern the developer is confined to writing as little code as possible for the application to function, de-
creasing overall development time and preventing feature tangents that are never actually utilized.

Finally, when application code is written, from the tests that were created. These tests were created from the acceptance tests, which were in turn created from a story. The story points to a feature. All of these steps seem excessive but because they all exist they insure that a clear pattern is followed to get from a feature to code. The developer then writes as little code as possible to get the test to pass.

Code Block 3.14

```ruby
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
```

This code will pass the test but is not very clean. The next step is to evaluate the code and refactor it. Refactoring is the process of changing to code to be more expressive and cleaner, without changing the underlying functionality ("The Rspec
Refactoring can take the form of pulling code from a method into its own methods and functions to improve readability. It can also take the form of just simplifying logic and taking advantage of other already existing methods. Lastly refactoring can take the form of renaming variables or removing them.

In the above code the first refactor and most obvious refactor is to rename the variable $x$ to something that explains what it is, `can_edit`. Still, when all the $x$s are replaced with `can_edit`s the code is hard to read and does not have one clear purpose. While refactoring, the tests should be run every time a change is made to insure that the original functionality of the method is not altered.

After several iterations of refactoring several steps can be made. Removing the local variable, simplifying logic, and ensuring a default return are all best practice and should be added during the refactor process.
The final result is a very concise method that is easy to read and modify during any further expansion of the code. After no further refactoring can be achieved the next feature is addressed and the process begins again.
Chapter 4: Results

This project focused on observing the effectiveness of BDD in a project driven by a group. The communication with the focus group highlighted many different strengths and weaknesses of BDD. By studying how the focus group grew over the semester to become more involved in the project as well as recording its input, a great deal of data was gained for each feature added to the project. Narwhal, starting as a very minimal application, grew to a nearly fully featured application in a matter of months using just one developer. A large part of this development speed can be attributed to BDD's focus on constant client communication.

A strength of BDD is how effective it is when working with the client through each individual feature. Rather than a project manager discussing a final product with the client, BDD emphasizes client communication through the development process. Pivotal Tracker helped Narwhal with this communication, providing feedback for each stage of feature development. The cycle of client communication can be expressed as a feedback loop, where both the developers and clients are commenting and providing feedback on each others work.

s.4.1: The Feedback Loop

The feedback loop is the process by which each feature in Narwhal is developed. The loop begins with a client, or in the case of Narwhal a member of the focus group, thinking of a feature they would like added to the current project. The cli-
ent then meets with the developers and discusses this feature, and out of this discussion comes a story. After the story is created, the developers begin creating the proper acceptance tests followed by Rspec tests and finally application code.

Through this process of creating each of these parts, it is the job of the developer to keep the client informed about feature development progress. Pivotal Tracker provides an excellent way to automate basic status feedback for the client. In Pivotal Tracker when a developer starts a feature, they must click **Start**, as shown in Figure 4.1.

![Figure 4.1](image)

After the developer clicks **Start**, the creator of the story is notified that development on one of their features has begun. By notifying the client that a feature has been started, the client remains involved in the development process and aware of the current status of their application. After the feature is completed, the developer must change the status of the current feature on Pivotal Tracker from **Start** to **Finished**, by clicking on the **Finish** button as shown in
Figure 4.2. This action notifies the client that a feature has been finished and will be ready for testing soon.

Then finally after the feature has been deployed to a place where the client can test or try the new code, the developer clicks Deliver as seen in Figure 4.3. The delivery notifies the client one last time, informing the client that the client may now test the feature. By clicking the deliver button two new buttons are also displayed, Accept and Reject, as shown in Figure 4.3.
These buttons are not for the developer to click but for the client. The Accept button is used if the feature meets all specifications and is bug free. If so, the client will accept the story and the feature will be migrated into production code, resulting in Figure 4.4 being displayed.

On the other hand if the client rejects the story for any reason, it is the job of the developer and the client to reconvene and discuss the reason for rejection.
Pivotal Tracker mandates this feedback loop by enforcing that each development stage is clicked through. Figure 4.6 demonstrates the feedback loop provided by Pivotal Tracker.
By enforcing this cycle of feedback the developer cannot follow a development tangent for a long time without the client being in the loop.

**s.4.2: Student Stories and Their Effects**

**s.4.2.1: Breaking Stories Down**

While in the process of developing Narwhal I observed a tendency of a story's creator to try and bloat their story after its creation. Stories in Pivotal Tracker must remain simple and clean, and should implement the smallest noticeable changes in a feature. Many times a story for Narwhal was rejected, not because it
was incorrect but because the user who created the story wanted to add additional specifications to the feature. It is the job of the developer to discuss the reason for rejection with the client and help them understand when a rejection should be moved to its own story. A good example of this is a feature, story number 18993917:

**Figure 4.7**

- As a presenter I should be able to disable/enable the canvas

This story is fairly clear, but it subsequently was rejected because its creator wanted clear notification about the current status of the drawing canvas, rather than just a simple widget saying that collaboration is on and off, because notification has nothing to do with the story 18993917 and the status of the canvas can change independently of notification, the rejection was modified, resulting in the creation and addition of a new story:

**Figure 4.8**

- I think the "Collaboration ON" indicator is placed kind of awkwardly. and I'm not sure if it's necessary at all.

This story begins to examine a new feature that would notify a user of the current on/off status of the canvas, which is treated as an additional feature. Sometimes there is a fine line between a rejection being a new feature or a modification of a previous feature. The developer and the client must discuss of the reason for rejection and come to an agreement about how to handle it. A simple rule told to the focus group was: only reject a story if it is not functional or does not meet the system requirements of the story, because it does not include components of the
story. By only rejecting stories that are do not meet the requirements, features are protected from repeat rejections that just bloat the original goal. Instead any extra requirements that could be added are put into a new story, creating more definable features.

About halfway though the semester, Pivotal Tracker added a Tasks feature which helps the developer and clients remain up to date on a feature’s current status. With the Tasks menu, as shown in Figure 4.9, acceptance tests for a story can then be checked off as completed, giving automated feedback to the client on just how far along a developer is in a current story. Although the task menu was not fully utilized for Narwhal because of its addition to Pivotal Tracker midway through the project, it still provided an excellent tool the stories for which it was actually was used.
s.4.2.2: Immediate gratification

By using Pivotal Tracker for Narwhal and following BDD, when a feature strayed from its original intent, the constant communication with the focus group resulted in immediate notification to the developer. It is a common problem for developers to try to solve a larger problem than the one at hand ("Introducing BDD"). By using Pivotal Trackers' Accept and Reject buttons, stories that strayed from the desire of the creator were reigned in quickly and effectively, with minimal damage or extra work needed. For example, during the development of Narwhal a story focused on visibility of presentations required a few revisions in order to be acceptable to the requester. The story read:
As a user, I’d like to see all available presentations.

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.

This story is a fairly vague and gave the developer some freedom when it came to actually making a lobby. This freedom is not a problem and is actually kept in check by BDD itself. As development on this feature continued, it was completed and presented to the focus group. The creator of the story rejected the feature for a couple reasons, as listed below:

- 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

I then developed changes and the feature was retested with a total overhaul and finally accepted by the creator. BDD thus prevented an unsatisfactory feature from becoming too buried into the software and difficult to remove. The assurance that any tangential development will be fixed before it tracks too far from the original goal is a great comfort that the client communication of BDD makes possible, resulting in potentially large savings in software maintenance costs.
s.4.2.3: Development from Start to Finish

It is important to see how each iteration of development tied into the overall look of the application as it grew from start to finish. This section will list a key story followed by a screenshot from each week of development.

Figure 4.13 Iteration 1
Figure 4.14 Iteration 2

- As a user I want to be able to draw on a canvas
Figure 4.15 Iteration 3

- Links should come in short form
Figure 4.16 Iteration 4

- As a user, I think the tool buttons should resemble other paint-type programs by using pictures for their icons as opposed to text.

Figure 4.17 Iteration 5

- As a user, I'd like to see all available presentations
Figure 4.18 Iteration 6

- Links should come in short form
- As a user, there should be some way to sync the resolutions of all the computers involved in the project. Someone was writing a long sentence in the project and the last two letters of each line were off of my screen.
As a user, there needs to be a way on the interface to easily join someone's class.
Figure 4.20 Iteration 8

- Previous story rejected
s.4.3 Shortcomings of Driving by Communication

While conducting this study, I did learn that BDD is extremely dependent on both the client and the developer being actively dedicated to the project. If either party is negligent in their responsibility to continue communication, the feedback loop breaks. Keeping the seminar class was a struggle in Narwal, because they were not generally interested in the development of Narwhal but instead were coerced with bonus points and grading. The coercion worked to some extent but enthusiasm toward seeing the application move forward is key to creating and updating stories. It was a regular occurrence in the meetings with the focus group, that they had to be primed on stories or features rather than inventing and creating entire features on their own. Having the original idea come from a de-
veloper rather than the client is a issue because the client will not have a clear picture of the feature the client is requesting and their judgment of the story is ultimately lacking original perspective. On Pivotal Tracker this can be seen with countless stories that either lack description or were fed to users by the developer.

Figure 4.22

- As a user, I would like an undo function.
  - https://www.pivotaltracker.com/story/show/19302973

- As a user, there needs to be a way on the interface to easily join some one's class.
  - https://www.pivotaltracker.com/story/show/19301757

- As a user, I would like to be able to have a shape tool
- In other words, a circle, square, rectangle, etc.
  - https://www.pivotaltracker.com/story/show/18994087

Without communication the development and progress of the application can come to an abrupt halt. Although Narwhal development never was halted, the focus group allowed features to stray from the original goal because of the lack of interest in the project as a whole. The lack of motivation was evident in the fact that stories would sit waiting for acceptance for weeks without any input from the users. The problem was eventually solved by educating the students about their importance in the development process. After giving some explanation about why direct and consistent input was a key component of the applications progress, members of the focus group seemed more interested in the application and the features they created.
Chapter 5: Conclusions

s.5.1: The Creation of Narwhal

The research project applied BDD to create a fully featured application by a single developer using a short timeline for development. After beginning development and following the rigorous guidelines of BDD, each small step in Narwal was completed quickly and easily. By the end of a four month development process, Narwhal moved from conception to application. A large portion of the success and speed of each feature’s creation can be attributed to the use of BDD.

As a single developer it was easy to get distracted. The weekly communication with the seminar class provided a grounding element for the project. Sometimes, weeks would pass without much progress because the feature I was working on were too large to finish in a single week and during these weeks it was difficult for the seminar class to provide guidance and additional features. However, their role was invaluable to my focus as a developer. A large percentage of software projects die before reaching deployment (Charette). Without the regular client communication with the seminar group it would have been likely that progress would slow, halt, or as developer I could have become diverted by a nonproductive development tangent.

There was a point during the development process where the seminar class and I had very different visions of approach. The problem that we were trying to address was the attempt to retrieve any of the presentations from the past. Nar-
whal users had previously either bookmarked their presentation, saved the URL in email, or remembered the presentation permalink. Saving the URL was a problem because people wanted to create presentations before they would actually present them, requiring Narwhal to recall past presentations created by each individual user. Initially this feature seemed simple and I completed it in a matter of days. However, while developing this new feature I took a short cut that resulted in its rejection by the client. If I had been unchecked by the client, the feature would have been undesirable, and have gone on to hinder further development, forcing me to go back and rework the feature after it had become embedded into the program.

The meetings with the seminar also provided motivation to continue development, even when feature production was not a priority and bug fixing was the most important task on the agenda. Bugs, just like features, were entered into Pivotal Tracker. Having to present my progress to a group of people, forced me to fix bugs quickly and efficiently rather than getting side tracked on features that were not a top priority to the seminar class. Bug tracking with Pivotal Tracker and using automated test suites provided an excellent safe-guard against code regression. Countless times I would refactor old code to be more functional with a new feature, thinking my changes left the older code functional. This was frequently true, but in some cases methods had to be changed to adapt to the new requirements. When the old methods were modified I had to step back through old code and make sure any calls to those methods could handle the refactored
versions. In a suite that grew as large as Narwhal, it became difficult for me to remember every occurrence of a single function call, and impossible for me to catch every error that arose from a refactor. The automated test suite written with Rspec and Capybara caught these errors for me. Allowing faster development and modification of old features for to adapt when to new requirements. The safety net that BDD provides around every turn allowed me to change code without the worry of tedious bugs arising and not being caught. Accessing bug reports in Pivotal Tracker and then following the BDD process with their development allowed me to identify and destroy bugs. Creating a story from a bug is done by first pinpointing the cause of the bug, breaking the problem into smaller pieces, correcting smaller pieces, and finally by automating and testing the bugs fixes. This process reduces the chance that the bug will arise again and the should it do so, the chance of it not being caught by the new.

s.5.2: Importance to Computer Science and Software Engineering

Studying software development is key to continued technology growth. BDD in particular is self correcting, it allows companies to produce much more advanced software at lower cost. Despite this, BDD is a hard sell. When I first approached the seminar class about the BDD, it was evident that initially the amount of work they were required to contribute to get a piece of software into production seemed burdensome. The work required by the client is also true for technology companies evaluating the cost of development against the value of the end prod-
uct. As the developer of Narwhal, I believe that the total development time of this program was reduced by strictly following the BDD process and guidelines. Although the extra work required writing tests and communicating development processes seemed like it required a developer to focus on things other than feature completion, each test the developer wrote, and each subsequent meeting with the client helped the developer create an outline of the next step. This outline, grounded in BDD, keeps developers from getting stuck in those moments when the next step is unclear. Utilizing BDD is like having road map. Reliance on this road map is an excellent tool for both fresh and seasoned developers. While developing Narwhal tests, small stories helped me to gain a better understanding of the immediate problem. There is a common saying in BDD when writing features, and trying to find a starting point, developers should be asking themselves the "5-why's" (Bulsuk). When approached with a feature a login page developer will say well

**Figure 5.1**

- I need to have 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 users work is only editable by them...

By continuing to deconstruct a feature the developer finds a clear starting point as along with identifying every part of the feature. For newer developers these requirements can help them split a larger problem into smaller ones. Then each solution will attain a visible success through testing. For older developers, BDD
assures that the most fundamental parts of the application are built first, then everything else is naturally built on top of this foundation, ensuring maintainability and adaptability without corrupting the bottom layer of the program.

The automated test suite does have disadvantages to new developers as they must learn to write a test as well as a code. This disadvantage is outweighed by the fact that using the automated test is a powerful way to clearly engage your code and identify bugs. This safety net provides new developers with a comfortable environment to experiment with their code, while assuring them that they won’t break something that was difficult for them to create initially.

s.5.3: Utility to Large Software Projects and Open source

A common problem in large software projects includes an "expert." Although the end result is good, having an “expert” forces a project to be reliant on one or a handful of people, and as those experts leave the project someone must fill their shoes. With extremely large projects loosing an “expert” is difficult because the loss of an expert can mean a halt in development until someone else can master the application’s code. By maintaining a strong test suite and enforcing the modular approach to features paramount to BDD, new developers can enter a project without having to understand the entire scope of the software they are editing. The test suite helps guide new developers and ensure that any changes they make cannot create bugs in other parts of the application. The modular approach also allows large teams to continue to create features inside a small box and only

61
modify the project's code when finally the feature is ready to be deployed into the final product. Should the deployment or any parts of the new feature affect the functionality of the software the tests will catch the errors and notify the developer before any harm is done.

These two parts of BDD also provide Open Source projects with a safety net to catch any code submitted to the main product. With the backing of communication and testing provided by BDD any developer, new or experienced, can contribute without causing fear to the project developers as new changes cannot harm the overall product. This safety net provides a clean way for a team of unrelated people to collaborate on a project with the confidence of knowing that their work cannot harm the software and that their advances will not be destroyed by future changes.

s.5.3.1: Improvements in the BDD process
BDD is a very clean and concise process that helps software engineers and their clients create software quickly and without worry. Currently it relies heavily on the involvement of two very dedicated parties in their product. BDD is constantly improving itself every day as more and more people adopt its procedures. Tools like Pivotal Tracker are a great way to minimize the amount of effort that is required by BDD. As these tools become available, they create a clearer vision of how a developer and client can communicate in order to drive development forward and minimizing disagreements.
Languages like Rspec and Capybara are excellent tools to bridging this gap and easing the stress of BDD. These languages focus on natural language and attempt to bridge the gap between client and developer. Although the languages are structured with natural definitions and a clean human-like syntax, they are still not simple enough to expose to a client. If the gap between client and developer can continue to shrink, by making testing language more accessible to non-programmers, less important information would get lost in the translation from client needs to feature code.

s.5.4: Future Work
The development of Narwhal helped demonstrate that BDD can aid in the development of a sturdy piece of software, control development tangents, keep developers producing features, and help developers tackle complex problems by breaking them into smaller pieces. This study was only the beginning of what can and should be studied about BDD and other development methodology. The next step in the study of BDD would be to repeat the same process, but with a client driven project rather than a group of volunteers. Using a client instead of a focus group would enforce communication with someone truly invested in the success of the project, and the advantages of BDD would be more clearly defined. Another future study would be the comparison of the results of BDD against competing software development methodology like TDD and simple iterations. By comparing various methodologies, researchers could draw conclusions about the
advantages and disadvantages of each method. A comparison study would enable clients and developers to choose the development style that best suits their need and the need of the project. The Narwhal project identified reasons why BDD should be considered as a viable software development process. BDD has the potential to help future developers start new projects correctly, or modify old projects to become safer, more reliable and more efficient.
Bibliography


