STAR WEST

SEPTEMBER 29–OCTOBER 3, 2008
DISNEYLAND® HOTEL
Anaheim,
California

T13

Concurrent Session
Thursday 10/02/2008 1:30 PM – 2:30 PM

# Driving Development with Tests: ATDD and TDD

**Presented by:**

**Elisabeth Hendrickson**
**Quality Tree Software, Inc.**

Presented at:
STAR*WEST* 2008
September 29 – October 3, 2008, Anaheim, CA, USA

## Elisabeth Hendrickson

**Elisabeth Hendrickson** has been in the software industry since 1984. Elisabeth has held positions as a tester, programmer, test automation manager, quality engineering director, and technical writer working for companies ranging from a twenty-person startup to a large multi-national software vendor. In 2003, Elisabeth became involved with the agile community, became a Certified ScrumMaster, and in 2006 joined the board of directors of the Agile Alliance. Today, Elisabeth spends her time teaching, speaking, writing, and working on Extreme Programming teams with test-infected programmers who value her obsession with testing.

# Driving Development with Tests: ATDD and TDD

## Presented at STARWest 2008

Elisabeth Hendrickson,
Quality Tree Software, Inc.
www.qualitytree.com
www.testobsessed.com

Instead of submitting slides for this session, I submit the following article describing the demo I will do.

## Driving Development with Tests

In Extreme Programming, programmers practice Test Driven Development (TDD). They begin developing code by writing a failing executable unit test that demonstrates the existing code base does not currently possess some capability. Once they have a failing unit test, they then write the production code to make the test pass. When the test is passing, they clean up the code, refactoring out duplication, making the source code more readable, and improving the design. TDD involves working in very small steps, one small unit-level test at a time.

Despite its name, TDD is a programming practice, not a testing technique. It happens to result in a fully automated suite of unit tests, but those unit tests are a side effect, not the ultimate goal. Practicing Test Driven Development is much more about setting concrete, detailed expectations in advance, and allowing those expectations of code behavior to guide the implementation than it is about testing.

Like TDD, Acceptance Test Driven Development (ATDD) also involves creating tests before code, and those tests represent expectations of behavior the software should have. In ATDD, the team creates one or more acceptance-level tests for a feature before beginning work on it. Typically these tests are discussed and captured when the team is working with the business stakeholder(s)[1] to understand a story on the backlog.

When captured in a format supported by a functional test automation framework like FIT or FITnesse, the developers can automate the tests by writing the supporting code ("fixtures") as they implement the feature. The acceptance tests then become like executable requirements.

[1] Here I use the term "business stakeholder(s)" to refer to the person or people with the responsibility and authority to specify the requirements for the software under development. In Scrum, this is the "Product Owner." In Extreme Programming this is the "Customer." In some organizations these people are Business Analysts; in other organizations they're part of Product Management. In each case, the business stakeholder specifies the "what" while the implementation team specifies the "how." The business stakeholder defines the features to be implemented in terms of externally verifiable behavior; the implementation team decides on the internal implementation details.

They provide feedback about how close the team is to "done," giving us a clear indication of progress toward a goal.

This paper explains the ATDD cycle in detail, providing examples of what ATDD and TDD tests look like at various points during the development process.

## Introducing the Sample Application

The sample application for this paper is a variation on a classic login example: it's a command-line based authentication server written in Ruby. At the moment, the sample application allows a user to do two things:

- Create an account with a password
- Log in with a valid user name and password

Attempting to log in with a non-existent user account or with an invalid password results in the same error message:

```
Access Denied
```

Creating a user account results in the message:

```
SUCCESS
```

And logging in successfully results in the message:

```
Logged In
```

So, for example, if I issue the following commands, I should get the following responses:

```
> login fred password
Access Denied
> create fred password
SUCCESS
> login fred password
Logged In
```

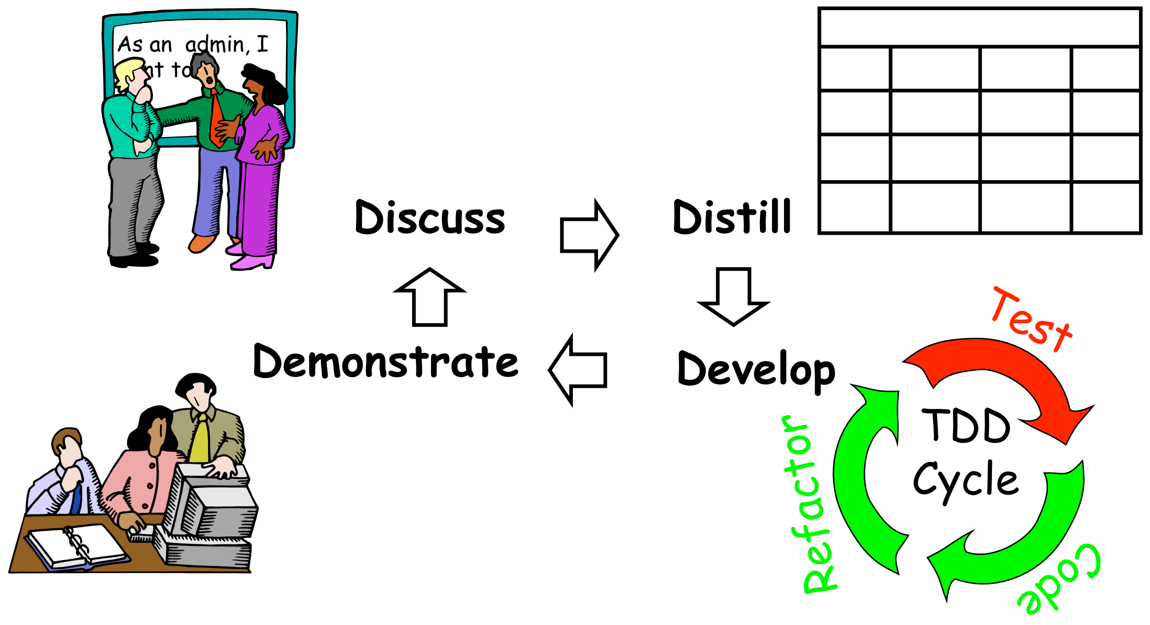
That's all our little sample application does just at the moment. It's not much. But it's a start. Now we need to add some new capabilities to it.

## The Backlog

In our sample project, the next story on the prioritized backlog is:

```
Users are required to use secure passwords (strings of at
least 6 characters with at least one letter, one number, and
one symbol)
```

## The Acceptance Test Driven Development (ATDD) Cycle



*ATDD cycle model by Jim Shore with changes suggested by Grigori Melnick, Brian Marick, and Elisabeth Hendrickson*

## ***Discuss*** the Requirements

During the Planning Meeting in which we discuss the story about secure passwords, we ask the business stakeholder requesting the feature questions intended to elicit acceptance criteria:

> *"What should happen if a user enters an insecure password?"*
>
> *"Can you give me examples of passwords you consider secure and insecure?"*
>
> *"What are examples of 'symbols'?"*
>
> *"What about spaces?"*
>
> *"What about dictionary words with obvious substitutions that meet the criteria but still may be insecure, like 'p@ssw0rd'?"*
>
> *"What about existing accounts?"*
>
> *"How will you know this feature 'works'?"*

During the discussion, we may discover that what looks like a simple little requirement opens up a can of worms. "Oh!" says the business stakeholder. "We should force existing account holders with insecure passwords to update their password on next log in."

By asking the right questions we are prompting the business stakeholder to think carefully about his expectations for the feature and gleaning insight into the acceptance criteria for the feature.

Sometimes we will decide as a team that an expectation is actually a separate story. "Let's make forcing existing account holders to update insecure passwords into a new story," we say. "It's really a separate feature." The business stakeholder agrees: "You're right. It's not as important as forcing new account holders to use secure passwords. It could be implemented later." This is a natural outcome of discussing the acceptance criteria for the feature in such detail, and saves us from having arguments over "scope creep" later on.

Once we have some understanding of what the business stakeholder expects the software to do, or not do, we can begin sketching out acceptance tests in collaboration with the business stakeholder. We do this in natural language. For example:

Valid passwords that should result in SUCCESS: "p@ssw0rd", "@@@000dd", "p@ss w0rd", "p@sw0d"

Invalid passwords that should result in the error message, "Passwords must be at least 6 characters long and contain at least one letter, one number, and one symbol.": "password", "p@ss3", "passw0rd", "p@ssword", "@@@000"

## ***Distill*** **Tests in a Framework-Friendly Format**

Now that we have the tests sketched out, we can capture the tests in a format that works with our test automation framework. There are a variety of test automation frameworks that support defining the tests in advance of the implementation including FIT, Fitnesse, Concordian, and Robot Framework. In this particular case, I am using Robot Framework.

Our Robot Framework tests will go into an HTML file that looks like this:

| Test Case | Action | Argument |
|---|---|---|
| Verify valid and invalid passwords | Password Should Be Valid | p@ssw0rd |
| | Password Should Be Valid | @@@000dd |
| | Password Should Be Valid | p@ss w0rd |
| | Password Should Be Invalid | password |
| | Password Should Be Invalid | p@ss3 |
| | Password Should Be Invalid | passw0rd |
| | Password Should Be Invalid | @@@000 |

Of course there are numerous ways we could express these tests, just as there are numerous ways we could capture ideas in prose. In distilling the tests into a table, we do our best to express the intent of the tests as clearly as possible without worrying (yet) about how these tests will be automated. That means we focus on the essence of the test and ignore details of implementation.

During the next part of the cycle, when we implement the software, we hook the tests up to the code.

## ***Develop*** **the Code (and Hook up the Tests)**

When implementing the code, if the developers are following a test-first approach, they execute the acceptance tests and watch them fail. In this case, the tests will fail with error messages like the following from Robot Framework:

```
No keyword with name 'Password Should be Valid' found
```

This is a perfectly reasonable failure given that we have not yet implemented this keyword in the framework. We expressed the tests the way we wanted to express them without worrying (yet) about automating them. So now it is time to think about automating the tests by writing the keywords that will connect the tests to the code.

With Robot Framework, that might mean creating the two keywords we need in library code. Similarly, with FIT and Fitnesse we would need to add code to a Fixture to hook the tests up to the code.

In this particular case, however, because we already have an existing system in place using Robot Framework, we already have automated keywords to create a new account and to log in. So we could rewrite the tests like so:

| Test Case | Action | Argument | Argument |
|---|---|---|---|
| Verify valid and invalid passwords | Create Login | fred | p@ssw0rd |
| | Message Should Be | SUCCESS | |
| | Attempt to Login with Credentials | fred | p@ssw0rd |
| | Message Should Be | Logged In | |
| | Create Login | fred | @@@000dd |
| | Message Should Be | SUCCESS | |
| | Attempt to Login with Credentials | fred | @@@000dd |
| | Message Should Be | SUCCESS | |
| | *...etc...* | | |

However, that results in long tests that obscure the essence of what we want to verify with a great deal of duplication. The original expression of the tests is better.

In Robot Framework, we can create new keywords based on the existing keywords, like this:

| Keyword | Action | Argument | Argument |
|---|---|---|---|
| Password Should Be Valid | [Arguments] | ${password} | |
| | Create Login | fred | ${password} |
| | Message Should Be | SUCCESS | |
| | Attempt to Login with Credentials | fred | ${password} |
| | Message Should Be | Logged In | |
| | | | |
| Password Should Be Invalid | [Arguments] | ${password} | |
| | Create Login | barney | ${password} |
| | Message Should Be | Passwords must be at least 6 characters long and contain at least one letter, one number, and one symbol. | |
| | Attempt to Login with Credentials | barney | ${password} |
| | Message Should Be | Access Denied | |

The important thing to note here is not the specific syntax that's particular to Robot Framework, but rather that we need to hook up the keywords used in the test to executable code. Different Agile-friendly test frameworks have different mechanisms for hooking up tests to code, but all of them provide some mechanism for hooking up the tests to the software under test.

Now that we have implemented the keywords, we can run the tests again and get much more meaningful results. Instead of an error message telling us that the keyword has not been implemented, we have tests that fail with messages like:

```
Expected status to be 'Passwords must be at least 6
characters long and contain at least one letter, one number,
and one symbol.' but was 'SUCCESS'
```

Progress! We now have a test that is failing because the software does not yet have the feature that we are working on implemented. Passwords can still be insecure. Of course, we knew the test would fail. We haven't done any work yet to make the test pass. However, we run the test anyway. There is always a chance that we implemented the test incorrectly and that it would pass even though the code is not yet written. Watching the test fail, and verifying that it is failing for the right reason, is one of the ways we test our tests.

## Implementing Code with TDD

Let's follow along as Jane, the developer, implements the feature to make the acceptance test pass.

First, Jane runs all the unit tests to ensure that the code meets all the existing expectations. Then she looks at the contents of the unit tests related to creating new password. In this particular code base, she finds that there are a bunch of unit tests with names like:

```
test_valid_returns_true_when_all_conventions_met

test_valid_returns_false_when_password_less_than_6_chars

test_valid_returns_false_when_password_missing_symbol

test_valid_returns_false_when_password_missing_letter

test_valid_returns_false_when_password_missing_number
```

"That's interesting," she thinks. "It looks like there is already code written to handle insecure passwords." The unit tests are already doing one of their jobs: documenting the existing behavior of the code base, like executable specifications.

Peering a little more deeply into the code, Jane discovers that although there is code written to determine whether a password is valid or not, that code isn't used. The "valid" method is never called. It's dead code.

"Ewww," Jane mutters to herself under her breath. "Unused code. Ugh."

Digging through the rest of the tests, Jane discovers tests related to creating user account with passwords. She spies this test:

```
def test_create_adds_new_user_and_returns_success
  assert !@auth.account_exists?("newacc")
  return_code = @auth.create("newacc", "d3f!lt")
  assert @auth.account_exists?("newacc")
  assert_equal :success, return_code
end
```

She notes that the test simply creates a new account with user name “newacc” and password “d3f!lt” then verifies that the create method returned “success” as the return code.

“I bet I’ll have a failing test if I assert that creating an account with an invalid password fails,” Jane says.

She writes the following test based on the original:

```
def test_create_user_fails_with_bad_password
  assert !@auth.account_exists?("newacc")
  return_code = @auth.create("newacc", "a")
  assert !@auth.account_exists?("newacc")
  assert_equal :invalid_password, return_code
end
```

Notice that in order to write this test, Jane had to make a design decision about what the “create” method should return when asked to create an account with an invalid password. Rather than making that decision when writing the production code, Jane made the decision while writing the unit tests. This is why TDD is more about design than testing.

Running the unit tests, Jane is satisfied to see that the new unit test fails. When called with the parameters “newacc” and “a,” the “create” method happily returns “success.”

Jane makes the necessary changes to the “create” method, using the tested but so far unused method that checks whether or not passwords are valid.

She then runs the acceptance tests again, but she still sees the error message:

```
Expected status to be 'Passwords must be at least 6
characters long and contain at least one letter, one number,
and one symbol.' but was 'SUCCESS'
```

“Right,” she says to herself. “I forgot to add the message to the messages table.” She adds the correct error message to the messages table and is happy to see that the acceptance test passes.

Next, since Jane and the other developers on the team practice continuous integration, she updates her local machine with the latest code from the source control system, manually merges any conflicting changes on her local development box, then runs all the unit tests again to make sure everything is green. When all the tests pass, she checks in the code changes.

## Exploratory Testing and ***Demo***

Jane has checked in the changes to the code to support the new feature, but she isn’t done yet. She works with others on the team to explore the feature. “Let’s try it with a password like ‘&-_\tf00’!” suggests Jack, another team member. Jack tries the command:

```
> create wilma &-_\tf00
```

The system responds with:

```
-bash: -_tf00: command not found
```

“Is that a bug?” Jack asks.

“Try it like this,” says Jane, “with quotes around the password,“ as she demonstrates:

```
> create wilma “&-_\tf00”
```

The system responds with:

```
SUCCESS
```

"If a user enters characters that have a special meaning to the UNIX shell, like "&", the shell will try to interpret them," Jane explains. "This is true anywhere in this system, not just on creating logins. Perhaps we need a story related to preventing users from entering such characters, or handling them differently?" She asks.

"I certainly think we should talk to the product owner about it," replies Jack.

As this example demonstrates, this kind of manual exploratory testing is essential to reveal holes in the acceptance criteria and discover risks that no one has thought about yet.[2]

Once the team is satisfied that the implementation matches the expectations, they demo the feature for the business stakeholder, bringing up the potential risks they noticed during implementation and exploration, like the "&" example above, and the discussions begin all over again.

## Results of ATDD

Teams that try ATDD usually find that just the act of defining acceptance tests while discussing requirements results in improved understanding. The ATDD tests force us to come to a concrete agreement about the exact behavior the software should exhibit.

Teams that follow the process all the way through, automating the tests as they implement the feature, typically find that the resulting software is more testable in general, so additional automated tests are relatively easy to add. Further, the resulting automated regression tests provide valuable, fast feedback about business-facing expectations.

## Acknowledgements

*The ideas in this paper have been heavily influenced by the following people, both in published work and in casual conversations. This paper would not have been possible without them.*

James Shore, Grigori Melnick, Brian Marick, Bas Vodde, Craig Larman, Lasse Koskela, Pekka Laukkanen, Juha Rantenan, Ran Nyman, Jennitta Andrea, Ward Cunningham, Rick Mugridge, Robert (Uncle Bob) Martin, Alex Chaffee, Rob Mee, Lisa Crispin, Kent Beck, David Astels, Antony Marcano, Joshua Kerievsky, Tracy Reppert, and all the members of the AA-FTT community I did not already list specifically by name.

[2] I wrote about using Exploratory Testing on Agile projects in a guest article published in *The Art of Agile Development* by James Shore and Shane Warden. I recommend reading that article for more details on how Exploratory Testing augments the automated acceptance tests. It is outside the scope of this presentation to discuss Exploratory Testing on Agile projects.

## Resources

Astels, David (2003). *Test-Driven Development: A Practical Guide*. Prentice Hall PTR.

Beck, Kent (2003). *Test-Driven Development: By Example.* Addison-Wesley.

Crispin, Lisa (2005). "Using Customer Tests to Drive Development." *Methods & Tools*. Summer 2005 Issue. Available online at http://www.methodsandtools.com/archive/archive.php?id=23

Koskela, Lasse (2007). *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications.

Marick, Brian (2003). "Agile Testing Directions." (An explanation of business-facing v. code-facing tests.) Available online at http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2

Mugridge, Rick and Cunningham, Ward (2005). *Fit for Developing Software: Framework for Integrated Tests*. Addison-Wesley.

Reppert, Tracy (2004). "Don't Just Break Software, Make Software: How storytest-driven development is changing the way QA, customers, and developers work." *Better Software Magazine*. July/August 2004 Issue. Available online at http://www.industriallogic.com/papers/storytest.pdf