The Triad,
A Tale of Lean-Agile Acceptance Test Driven Development
July 2010

Ken Pugh

Fellow Consultant
Net Objectives
ken.pugh@netobjectives.com

0

Front Matter

Dedication

I'd like to dedicate this book to three people. My brother Bob inspired me to become an engineer. I recall one time when he was home from college and presented me with the N-body problem [Wiki01] and the four color map problem[Wiki02] My science high school teacher, Mr. Sanderson, spurred me on to explore topics as why there is air[1]. My mechanical engineering professor at Duke, Dr. George Pearsall, encouraged exploration. In his strength of materials class, I discovered why my guitar strings broke.

Preface

Developing software with testable requirements is the theme of this book. A testable requirement is one with an acceptance test. The acceptance tests drive the development of the software. As many development groups have experienced, creating acceptance tests prior to implementing requirements decreases defects and improves productivity. A triad - the customer / business analyst, developer, and tester collaborate on producing tests to help create a high quality product. Acceptance test driven development is as much about specifying what is to be done as it is about testing the implementation.

You are starting to read this book. Do you have criteria in mind as to whether the book will met your needs? If you finish this book, how will you know whether it has met those criteria? This book represents an implementation of something that should help meet your needs. Since you are reading this after the book has been finished, you don't have an opportunity to influence the acceptance criteria. So let me list them here and see if this is what you are after:

From my days in English classes, the teacher emphasized that a story should contain a who, what, when, where, why, and how. So I've made that the goal of this book. It explains:

- Who should create acceptance tests
- What are acceptance tests
- When should the acceptance tests be created
- Where are the acceptance tests and requirements stored
- Why – the benefits of having acceptance tests

[1] And not just Bill Cosby's answer.

- How the acceptance tests are created

As a start to answering these questions, the title of this book - The Triad -- refers to power of the three [Crispin01] who creates the acceptance tests -- the customer, developer, and tester One definition of an acceptance test is "a test performed by the end user to determine if the system is working according to the specification in the contract." [Answers01] By the end of the book, the expectation is that you should understand how testable requirements can make the software development process more enjoyable (or at least less painful) and can help in producing higher quality products.

Why ATDD

Let's start with the answer to the why question. That may give you motivation to read the book to get the answers to the other questions. Jeff Sutherland, the co-creator of Scrum, has metrics on software productivity. [Sutherland01]. He found that adding a quality assurance person to the team and creating acceptance tests prior to implementation doubled the productivity of the team. Your actual results may vary, but teams adopting acceptance test driven development have experienced productivity and quality increases. Mary Poppendieck says that creating tests before writing code is one of the two most effective process changes that can produce quality code faster.[2] [Poppendieck01]. Customer-developer-tester collaboration reduces unnecessary loops in the development process. As Michael Bolton says, "Tri-over avoids try-over". As Jerry Weinberg and Don Gause wrote, "Surprising, to some people, one of the most effective ways of testing requirements is with test cases very much like those for testing the completed system." [Weinberg01]

If you are going to test something and document those tests, it costs no more to document the tests up front than it does to document them at the end. But the tests are more than just tests. As stated in Chapter 3, "Requirements and tests are linked together. You can't have one without the other. … The tests clarify and amplify the requirements. A test that fails shows that the system does not properly implement a requirement. A test that passes is a specification of how the system works..."

Acceptance tests can be manual tests. But if they are automated, you can use them as regression tests to ensure that future changes to the system do not affect previously implemented requirements.

As you will see, acceptance tests are a measure of the complexity of a story. If a story has many tests, it may be broken up. The number of passing acceptance tests is a relative measure of progress. When all acceptance tests are passed, a story is done.

What types of software does this cover?

The acceptance tests we cover in this book revolve mainly around requirements which have results that can be determined. These results are typical in business situations. You place an order. The order total is determinable. On the other hand, you have a requirement to find the shortest possible path that goes through a number of points. This is the traveling salesman problem. For a small number of points, the result is determinable by brute force. However for a large number of points, the answer is not determinable. You can have a test that checks the output of one way of solving the problem against the output of another way. But that does not guarantee that the shortest solution has been found.

Having acceptance tests may not help the developer create an algorithm. For Sudoku puzzles, the answer is determinable. You can have an input puzzle and an output answer. So you can easily check whether a particular algorithm has correctly solved the problem. But there is no guidance from the test as to how to find that algorithm.

[2] The other is quick feedback.

How will we get there?

A continuous example throughout the book shows the steps involved. Some steps are less detailed than others. There are books devoted entirely to those steps and reference is made to the material which has much greater explanation. In particular, the appendix gives references for tools to program the acceptance tests, to the agile process itself, to requirement elicitation, and to testing the other qualities of a software system (usability, performance, etc.).

The material is presented in three parts. The first part documents the tale of the triad as they develop a software system. It shows how acceptance testing permeates the entire process, from the project charter to individual stories. The second part covers particular issues in the process, including how it works with larger projects. The third portion includes case studies from real-life situations. In some cases the studies have been simplified to show only the relevant parts.

The example of Sam's CD Rental Store follows along Sam's story in *Prefactoring – Extreme Abstraction, Extreme Separation, Extreme Readability*. That book used the story as the context for examples of good design. *Prefactoring* covered some of the aspects of developer-customer interaction, since a good design requires understanding the customer's needs. *Prefactoring*'s focus was on the internal software quality. This book's focus is on delivered quality. So the two books complement each other.

Lineage

A Chinese proverb says, "There are many paths to the top of the mountain, but the view is always the same." And many of the paths share the same trail for portions of the journey. Although acceptance testing has been around for a long time, it was reinvigorated by Extreme Programming[Jefferies01] [ref]. Its manifestations in

Acceptance Test Driven Development (ATDD), Example Driven Development (EDD) by Brian Marick [Marick01], Behavior Driven Development (BDD) by Dan North [Chelimsky01], Story- Test Driven Development (STDD) by Joshua Kerievsky of Industrial Logic [Kerievsky01], Domain Driven Design (DDD) by Eric Evans Evans01] all share the common goal of helping to produce high-quality software. They help developers and testers to better understand the customer's needs prior to implementation and for the customer to be able to converse in their own domain language.

Many aspects are shared between the different approaches. ATDD in this book encompasses many aspects of the other approaches. I've documented the parts that come specifically from the other DDs, including Brian Marick's examples, Eric Evan's ubiquitous language, and Dan North's given-when-then template. The most visible differences areis that the tests here are presented in table format, rather than in a more textual format, such as BDD's Cucumber language, and they concentrate on functionality, rather than user interface. This book's version of ATDD matches closely that described by Lasse Koskela . [Koskela01] and follows the recommendations for testing of Jim Coplien [Coplien01]..

One of the most well known DDs is Test Driven Development (TDD) by Kent Beck [Beck01]. TDD encompasses the developer's domain and tests the units or modules that comprise a system. TDD has the same quality goal as ATDD. The two interrelate since the acceptance tests can form a context in which to derive the tests for the units. TDD helps in creating the best design that meets the acceptance tests. Design issues include assigning responsibilities to particular modules or classes to pass all or part of an acceptance test.

Automation

I emphasize acceptance tests as customer-developer-tester communication. There is little information about frameworks for automating acceptance tests. If you don't have an acceptance test, there is nothing to automate.

I do not advocate any particular test automation framework. The book does suggest that table-driven tests can be easy to create. These tables follow along the concepts of David Parnas who states "The tables constitute a precise requirements document." [Parnas01]

The tests can be run manually against a user interface. However the most effective use of them is to turn them into an executable specification. To demonstrate automation, an example in the appendix uses the Fit framework developed by Ward Cunningham and James Shore. The code for the example is available on-line.

Acknowledgements

Over my two-fifths of a century in software, I've have the opportunity to interact with a wide range of people. Many of the ideas expressed in this book have come from these people - from their books, their talks, and personal conversations. Albert Einstein said, "Creativity is knowing how to hide your sources." I would like not to hide these people.

Here are others who have influenced the ideas in this book in no particular order: The only problem is I can't always remember what I got from whom. The list includes in no particular order: Cem Kaner, Jerry Weinberg, James Bach, Michael Bolton, Brian Marick, Ellen Gottesdiener, Karl Wiegers Ward Cunningham, Jim Shore, Rick Mugridge, Lisa Crispin, Janet Gregory, Kent Beck, Gerard Meszaros, Alistair Cockburn, Andy Hunt, Bob Martin, Dale Emery, III, Michael Feathers, Mike Cohn, Jim Highsmith, Linda Rising, Ron Jeffries, Mary Poppendieck, Jim Coplein, Norm Kerth, Scott Ambler, Jared Richardson, Dave Thomas, Martin Fowler, Bill Wake, Tim Lister, Eric Evans, Bret Pettichord, Brian Lawrence, Jeff Patton, David Hussman, Rebecca Wirfs-Brock, Joshua Kerievsky, Laurie Williams, Don Gause, James Grenning, Tom DeMarco, Danny Faught, Jeff Sutherland, David Astels, Lee Copeland, Elisabeth Hendrickson, Bob Galen, Gary Evans, George Dinwiddie, Jutta Eckstein, Bob Hartman, David Chelimsky, Dan North, Lasse Koskela, Cedric Beust, and Larry Constantine.

I'd like to thank Rob Walsh of Envisionware for the case study of a library print server.

I also thank my Net Objectives gang, Alan Shalloway, Jim Trott, Scott Bain, Amir Kolsky, Cory Foy, and Alan Chedalawada.

In helping make this book a reality, I thank the people at Addison-Wesley Pearson Technical Group - Chris Guzikowski, Chris Zahn, Raina Chrobak, and … And to reviewers Andrew Binstock, SGuy GeXiang Ge, Tom Wessel, Kody Shepler, Jinny Batterson, (** Add **).

And last but not least, I thank Leslie Killeen, my wife. She is a weaver. Software is not her field. She reviewed my drafts, gave helpful hints, and supported me through the creation process.

Acceptance test driven development:
The answer is 42. Now implement it.

0A

Outline

Contents

Part One – The Tale

This part tells the tale of a team developing a project. It starts with an exploration of testing. The team develops a charter followed by a set of high level requirements. Requirements are broken down into stories. Stories are detailed with use cases and business rules. Acceptance tests are developed for every business rule and every scenario.

Chapter 1

Prolog

“"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop”

Lewis Carroll, *Alice's Adventures in Wonderland*

Say hello to testable requirements. You get an introduction to what acceptance tests are and the benefits of using them. You are introduced to the team that will be creating them.

The “As-Is” Situation

It’s four o’clock on the last day of the iteration. Tom, the quality assurance tester is testing the implementation that the developers handed over to him at three-thirty. He goes through the screens, entering the test case data he created that afternoon. Clicking through the screens, it looks like everything is correct. He signs off on the requirement. At the demo the next day, Cathy, the customer looks at the results and rejects them. The numbers aren’t right. One field should not have shown up since the account was inactive. That’s one more not-done requirement that makes Cathy unhappy. The requirement gets scheduled again in the next iteration to fix the issues.

It’s now five o’clock on the last day of the next iteration. Tom is hard at work testing the stuff finished by the developers at four-thirty. Going through the screens he discovers that the results aren’t what he had determined they should be. It is too late to contact Sam for clarification as to whether he made the correct assumptions. The developers have already gone home for the weekend. Nothing left to do but write up a defect to be addressed during the next iteration, leaving less time to develop new features.

The “To-Be” Situation

It’s four o’clock on the last day of the iteration. Tom is starting to test the story that the developers handed over to him at three-thirty. Debbie, the developer, has already run through the acceptance tests that Cathy, Debbie, and Tom created prior to Debbie starting implementation. Tom quickly runs through the same acceptance tests and then starts doing more testing to get a feeling for how the story results fit into the entire workflow. At the review the next morning, Sam agrees that the story is complete.

What’s the difference between the “as-is” and the “to-be”? Every requirement story in the “to-be” situation has one or more tests associated with it, making each a testable requirement. The tests were developed by the customer, tester, and developer prior to implementation. As we will see in later chapters with detailed examples, those tests clarify the requirements. They provide a measure of “doneness” to all of the parties. In the “as-is” case, there were no tests created up front. The developer had nothing to test against, thus relied on the tester to perform validation. Feedback as to success or failure of a story was delayed.

If a requirement does not have a test, then it is not a testable requirement. If you cannot test that a requirement has been fulfilled, how do you know when it has been met? This does not mean that the test

is easy to perform. But there must be at least an objective test so that the customer, developer, and tester have a common understanding of what meeting the requirement implies.

A Car Story

In my classes, I often start with a dialog to emphasize the importance of acceptance criteria. It usually goes something like this:

I ask, "Does anyone want a fast car?"

Someone always says, "Yes, I want one."

"I'll build you one," I reply. I turn around and work furiously for five seconds. I turn back around and show them the results. "Here's your car", I state.

"Great," they answer.

"It's really fast. It goes from 0 to 60 in less than 20 seconds," I proudly explain.

"That's not fast," they retort.

"I thought it was fast. So give me a test for how fast you want the car to be.", I reply.

"0 to 60 in less than 4.5 seconds," they state.

I turn back around, again work quickly, and then face them again. "Here it is 0 to 60 in 4.5 seconds. Fast enough?" I ask.

"Yes," they answer.

"Oh, by the way, 60 is the top speed," I state.

"That's not fast," they retort.

"So give me another test," I ask.

"Top speed should be 150," they demand.

Again, I quickly create a new car. "Okay, here it is 0 to 60 in 4.5 seconds. Top speed of 150. Fast enough?" I again ask.

"That should be good," they retort.

"Oh, by the way, it takes two minutes to get to 150," I let slide.

By this time, my point has been made. Getting just a requirement for a "fast car" is not sufficient to know what to build. The customer needs to create some tests for that requirement that clarify what exactly is meant by "fast". Without those tests, the engineers may go off and create something that they think meets the requirement. When they deliver it to the customer, the customer has a negative reaction to the creation. The item does not meet his needs, as he thought he had clearly stated in the requirement.

Having acceptance tests for a requirement gives the developers a definitive standard against which to measure their implementation. If the implementation does not meet the tests, they do not have to bother the customer with something that is non-compliant. If each acceptance test represents a similar effort in creating the implementation, then the number of passing tests can be used as a rough indication of how much progress has been made on a project.

Although the tests are absolute, such as "0 to 60 in less than 4.5 seconds", they also form a point of discussion between the customer and the engineers. If the engineers work for a while and the car accelerates in 4.6 seconds, they can discuss with the customer whether this is sufficient. In particular, this would occur when the engineers discover that getting the time down to 4.5 seconds might take considerable more development time. In the end, the customer is the decision maker. If 4.5 seconds is absolutely important to sell the car, then the extra development cost is worth it. If not, then money will be saved.

I'd like to clarify a couple of terms that are used throughout the book - acceptance criteria and acceptance tests. Acceptance criteria are general conditions of acceptance without giving specifics. For the car example, these might be "acceleration from one speed to another", "top speed", and "must feel fast". Acceptance tests are specific conditions of acceptance, such as "0 to 60 in less than 4.5 seconds". Each acceptance test has unstated conditions. For example, an unstated condition could be that the

acceleration is measured on a flat area, with little wind. You could be very specific about these conditions: an area that has less than .1 degree of slope and wind less than 1 mile per hour. If necessary for regulatory or other purposes, you could add these to the test. But you should avoid making the test a complex document.

A few facets that differentiate acceptance tests from other types of tests, such as unit tests, are:

- Acceptance tests are understood and specified by the customer
- Acceptance tests do not change even if the implementation changes

Team Introduction

The principles and practices of acceptance test driven development are introduced through the story of the development of a software system. The story originated in my book, *Prefactoring – Extreme Abstraction, Extreme Separation, Extreme Readability* (Sebastopol, Calif.: O'Reilly, 2005). In that book, the emphasis was on how developers could develop a high quality solution for the system. This book highlights the customer-developer-tester interaction in creating and using acceptance tests.

The People and the System

Sam, the owner of Sam's Lawn Mower Repair and CD Rental Store, represents a project sponsor for a new system. His wife Cathy plays the role of customer – the one requesting the application. Cathy takes care of the logistic side of the business. She does the bookkeeping, handles the inventory, and places orders for new CDs. Other stakeholders include Sam's sister Mary and his brother-in-law Harry, who frequently help out at the store. Their son Cary is a clerk at the store. He will be working with the system that is developed.

Debbie is the developer. She appears as a single developer, rather than as part of a pair to keep the story simple. Tom is the quality assurance. I refer to him as a tester, but his responsibilities really include helping everyone create a quality product. Tom and Debbie work together as a pair to understand and implement what Cathy needs. The entire team is shown in Figure 1.1.

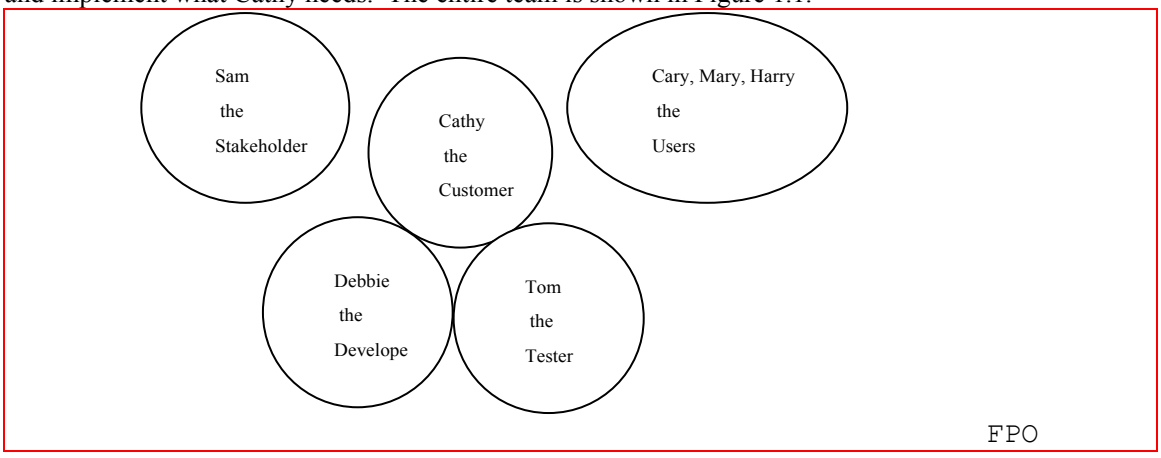


Figure 1.1 *The Team*

Sam had a lawn mower repair shop for a number of years. He noticed that people coming into the shop had circular devices hanging on their bodies. It turned out they were Sony Discmans. Being the

inquisitive type, he discovered that his customers liked to listen to music while they mowed the lawn. So he added CD rental to the services his store offered.

Business has been booming, even though the Sony Discman is not longer being used. People are now coming in with little rectangular boxes hanging around their necks or sitting in their pockets. They are renting more CDs than ever and returning them quickly – in as little as an hour. Sam's paper system is having a hard time keeping up and it is becoming difficult for Cathy to produce the reports needed to keep track of the inventory. Sam is planning on opening up a second store. Before he does that, he figures that he needs to create a software system or else the issues will just double.

He got a recommendation and called Debbie. Tom is her partner in a small development shop. They had great initial meeting with Sam and Cathy and were selected to implement the system. Along the way as you'll see, Cathy gets introduced to software development and in particular to acceptance test-driven development.

Summary

- A testable requirement has one or more acceptance tests
- An acceptance test is a standard to measure the correctness of the implementation
- Acceptance tests are created in collaboration with the customer

Chapter 2

Lean and Agile

"You're a lean, mean, fighting machine!"

Bill Murray as John Winger in *Stripes*

The triad has its first meeting. Debbie describes the differences between traditional development and acceptance test driven development (ATDD). Tom explains how ATDD is lean.

The Units and the Triad

Many agile books refer to the developer team and the business team. Names often have a connotation. On a football field, what do two teams do? They compete, to see who can score more points and win the game. On a single team, there is an offensive unit, a defensive unit, and a special teams unit. The offensive's unit job is to score points. The defensive's unit responsibility is to keep the other team from scoring. The special team unit has the goal of scoring points when it receives a kicked ball and preventing the other team from scoring when it kicks the ball. All three units must do their job to win the game.

Although each unit has a primary job, it doesn't stop there. If the offensive unit fumbles the football and the other team recovers, it does not simply stop playing and call for the defensive unit to come on the field. Instead, it has to play defense until the end of the play.

So the three units in a software project are the customer unit, the developer unit, and the quality assurance unit. The customer (who may have the title of product owner, business analysts) determines the requirements, creates acceptance tests, and sets priorities. The developer unit implements the requirements and ensures implementation meets the acceptance tests. Quality assurance helps both the customer unit to develop acceptance tests and the developer unit to pass those tests. The quality assurance unit is responsible for ensuring that the overall process produces high quality products. Part of that effort involves creating tests and testing. Since that is the primary effort to be discussed in this book, they will be referred to as the testing unit. The triad – the customer unit, the developer unit, and the testing unit – work together to produce quality software.

We start the story where Debbie, representing the developer unit, is explaining to Cathy, the customer unit, alternatives for software development. Tom, the testing unit, is sitting in the meeting of the triad (see Figure 2.1).

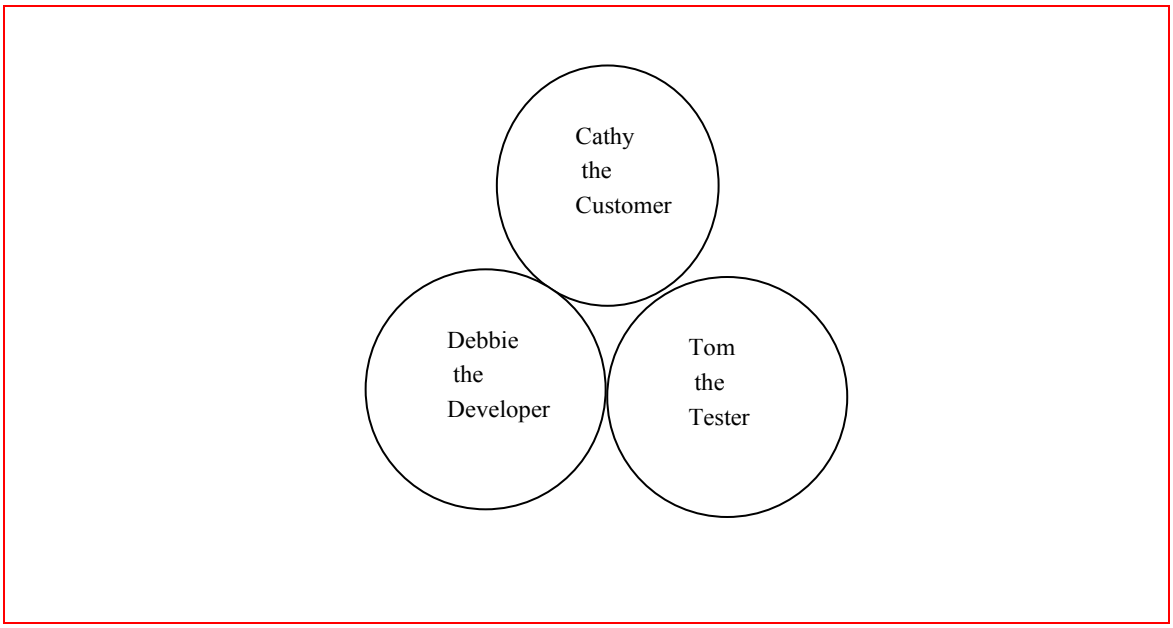


Figure 2.1 *The Triad*

The Process

Debbie and Tom meet with Cathy to explain how the development process works. Debbie starts off with how the flow often works and compares it to how Debbie and Tom prefer to work.

Debbie introduces the chart shown in Figure 2.2.

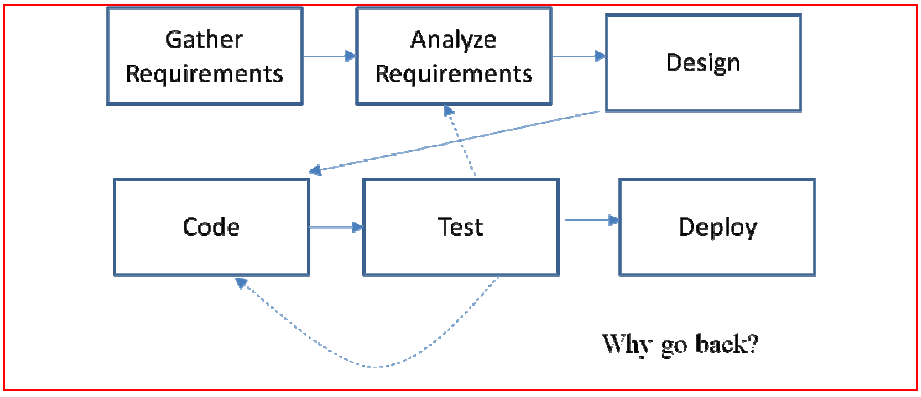


Figure 2.2 *Typical Development Flow*

(*** Gather should be Elicit in this diagram ***)

"Cathy, this is how development often looks for other teams", Debbie explained. "You start by eliciting some requirements. Then you analyze the requirements to see if they are consistent and understandable. Based on the analysis, you create a design for how you are going to implement the requirements. Then someone like myself would code the requirements. The program that I've coded is turned over to testers like Tom. Tom reads the requirements and then creates some functional tests that

they can run to see if the program meets those requirements. He runs the tests and sees whether the program produces the expected results."

"If the functional tests pass, as well as other tests, such as performance and usability, then the system is ready to be deployed. Now if everything is perfect, then the program passes through these stages in a straight line. But perfection only occurs in fairy tales. In reality, there are often misunderstandings. We don't always use the same words with the same meaning. You may say 'always', when you really mean 'usually'. Or I may hear 'usually' and think it means 'always'."

"Now when a misunderstanding is found, it needs to be corrected. If Tom finds that misunderstanding during testing, then we have to figure out how to correct it. It could be that I simply made a mistake in coding. So you see the line from test back to code in Figure 2.2. Tom tells me about the issue he discovered and I correct it."

"It could be a misunderstanding that occurred during requirement analysis. In that case, Tom and I would revisit the requirement. He might have interpreted one way and I might have interpreted another. We would check back with you to see which one of our interpretations was correct, or whether you meant something else entirely different."

"This cycling back causes delay in deploying the product, as well as extra cost in creating the product. So Tom and I like to operate in a different mode that uses quick feedback."

Quick Feedback Better than Slow Feedback

Before describing Debbie and Tom's alternative process, let's take a look at the idea of feedback. Feedback involves using an output to influence output in the future. Feedback in software development is not quite the same as feedback in control systems. In control systems, values from the output are feed back into the input. In audio systems, the output sound from the speakers can get back into the input to the microphone. That positive feedback can cause an explosive sound to emit from the speakers. Instead, think of software feedback as a listener commenting that the sound from the speakers is too loud. You adjust the amplifier volume to make the sound closer to what the listener likes.

Imagine that a listener tells you that he would like to hear your new stereo. He says he cannot hear the music very well. You slowly increase the volume. If the listener gives you frequent feedback, you'll stop the increase just after you've increased past where he wanted it to be. So the volume will be pretty close to what he wants. You may hone in even further by reducing the volume even more slowly. If the listener does not let you know frequently, then you will increase the volume well above what he wants and then decrease it well below. You will continually cycle between too loud and too quiet.

Quick Feedback on Mileage

Quick feedback promotes different processes. Energy efficient cars, such as the Toyota Prius and the Honda Insight, have instantaneous mileage displays. When you drive one of these cars, you find out quickly which actions you perform decrease or increase mileage. For example, rapid acceleration shows up as really low mileage. Trying to beat the car next to you when the light turns green quickly gives an indication that you used a lot of gas. Some people check their mileage every time they fill up the tank. But then it is hard to determine what actions during the previous tank-full caused either good or bad mileage. All that you can ascertain is that something during that period caused the mileage to vary.

Quick feedback means less delay. Quick feedback is good. The output will be closer to the desired outcome.

Quick Feedback on Data Entry

Here is another example of quick feedback. Suppose an entry screen has a credit card field. The format of the credit card number can be checked as soon as it is entered. Whether the card number is valid can be checked when the entry screen fields have sufficient information for validation (e.g. name, zip code, and so forth). When a charge is actually made, the transaction can be performed immediately, rather than being delayed by having incorrectly formatted input

An Alternative Process

Debbie, Tom, and Cathy take a break for a few minutes. Then Debbie starts to describe an alternative process. She shows Cathy the chart in Figure 2.3.

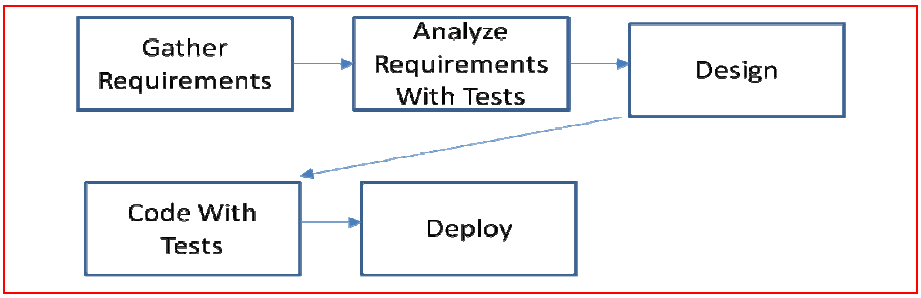


Figure 2.3 *Acceptance Test Driven Development Flow*

(*** Gather should be Elicit in this diagram ***)

"Here's the alternative process that Tom and I use. As we elicit some requirements, we sit down with you Cathy to work out some acceptance tests for the requirements. These are specific examples of the requirement in action. When I'm coding, I'll use these tests to ensure that my implementation meets the tests. When it does, I'll turn it over to Tom for the other types of tests that are discussed in the next chapter."

Tom chimes in. "This is a small application relative to some we've worked on in the past. Debbie's machine will be almost an exact duplicate of your computer on which the application will be deployed. So when she turns it over to me and I run it on the test system that exactly matches your computer, I don't expect that any tests that we've created should fail."

"He's right," Debbie agrees. "On larger systems, the environment on my machine in which the program is developed can be dramatically different than that on which it is run for deployment (see Figure 2.4). We'll talk more about that in the next chapter. But in this case, I'll be running the acceptance tests on my machine. When Tom runs them, they should all pass."

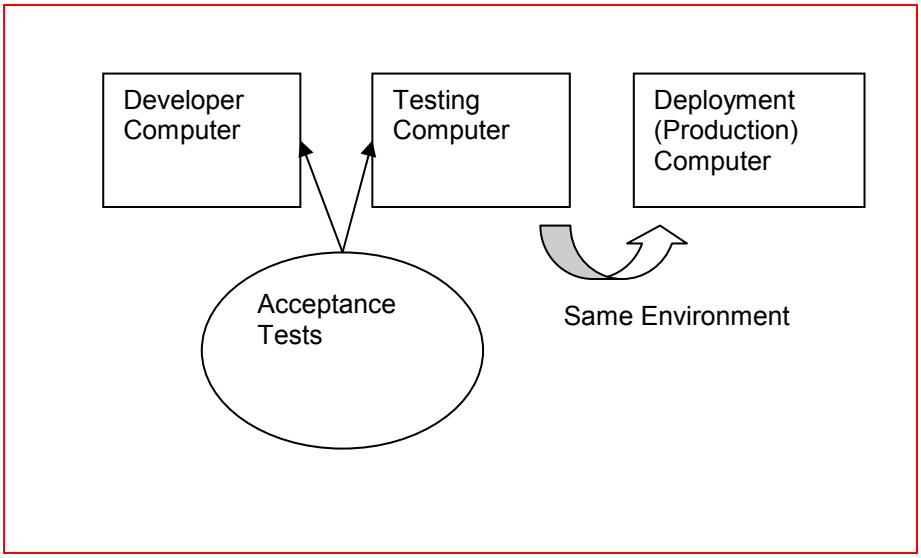


Figure 2.4 *Acceptance Tests*

Tom interjects, "Of course, since there's no sense in my running a test that failed on your machine."

"What this means", Debbie continues, "is that the three of us will be creating these tests together to make sure that we all have a clear understanding of what a requirements means. When a requirement is delivered, it will work as understood."

"Now we will not be doing this up front for all of the requirements. As you decide what the next requirement to work on is, we'll get together and get the details. I know you have to work at the store and there is no room for us to set up our computers at the store. But since we're only a couple of miles away and there is a great bike path between our places, we'll ride on over for these meetings. We've found that meeting face-to-face is much more effective than having a long conference call on the phone. [Cockburn01]

"If it's a quick question, we'll give you a call. That's not as ideal as being able to pop our heads up and ask a question face-to-face, but given the physical limitations, it will work out. And it's closer to the way things work in the real world than not. The only thing we ask if that you get back to us relatively soon. We can often work on other pieces of the problem, but if the question regards something that is fundamental to the design, we'll need a quick answer. And we'll identify the difference between that which we need as soon as possible and that which can wait a little while. Of course, if the little while turns into a day or a week, your project will be delayed and there will be more costs."

Lean and Agile Principles

Tom starts explaining to Cathy that the acceptance test-driven process is based on some lean principles and some agile principles. The lean principles come from Mary and Tom Poppendieck [Poppendieck02], [Poppendieck03], [Poppendieck04] The agile principles are in The Agile Manifesto, which is an widely-recognized statement on how to better develop software. [Agile01]

Tom starts off, "Mary and Tom developed principles of lean software development derived from lean manufacturing. Their book and Mary's papers on the web gives a much longer explanation than I'm going to give you. One principle is to eliminate waste from the process. Creating a requirements document which is 'thrown over the wall' to the developer and tester can create loopbacks in the process. Getting rid of these loopbacks by having agreed-upon acceptance tests eliminates waste."

"Another principle is to build in integrity. Acceptance tests for each portion of the system help to ensure that the modules are of high quality. The tests are run as each module is developed, not when the developers have completed the entire system."

"The collaboration between the three of us – the triad- helps amplify learning, another lean principle. We learn from each other about the business domain and the development and testing issues."

"Our triad is one manifestation of the Agile Manifesto principle that 'Business people and developers must work together daily throughout the project.' Although we may not be physically together, we will be working together continuously."

"Another agile principle is that 'Working software is the primary measure of progress.' With the acceptance tests, Debbie and I can deliver not only working software, but software that delivers more precisely what you are asking for."

Summary

- The triad consists of the three units – customer, developer, and tester – that collaborate to create high quality software
- Quick feedback is better than slow feedback
- Acceptance test driven development reduces unnecessary loopbacks
- ATDD is lean and agile

Chapter 3

Testing Strategy

"How do I test thee? Let me count the ways."

Elizabeth Barrett Browning (altered)

The triad is having a lunch time get together. Tom explains to Cathy about the different types of testing that occur during development. The acceptance tests that Cathy provides are only one part of the testing process. He also describes when and where the tests are run.

The Testing Matrix

Tom starts off "Cathy, acceptance tests are one part of the testing strategy for a program. I think the easiest way to describe the full set of tests is to use the testing matrix from Gerard Meszaros. [Meszaros01] The matrix in Figure 3.1 shows how acceptance tests fit into the overall picture.

(*** Use the image from that book ***)

<table>
<tr><td>Automated or Manual</td><td colspan="2">Business Facing</td><td>Manual</td></tr>
<tr><td rowspan="3">Functional</td><td>Acceptance Tests</td><td>Usability Tests</td><td rowspan="3">Critique Product
Cross-Functional</td></tr>
<tr><td>Component and Integration tests</td><td>Exploratory Tests</td></tr>
<tr><td>Unit tests</td><td>(Ilities) Performance, Security</td></tr>
<tr><td>Automated</td><td colspan="2">Technology Facing</td><td>Manual or Tools</td></tr>
</table>

Figure 3.1 *The Testing Matrix*

Tom continues, "*Acceptance tests* encompass the business facing functional tests that ensure the product is delivering the results that the customer wants. Almost every acceptance test can be expressed in yes or no terms. Examples are 'when a customer places an order of $100, does the system give a 5% discount?' and 'is the edit button disabled if the account is inactive?'"

"Acceptance tests are closely related to system test. System tests ensure that a software system meets the requirements. [Answers02] Both are black box tests, that is, they are independent of the implementation. Acceptance tests are often called user acceptance tests, since the end user defines them. System tests are

usually written by testers who read the requirements and create tests for them. If the triad writes the tests together, then the distinction between acceptance tests and system tests is practically eliminated"

"Another form of black box testing is functional testing. This checks that a particular function works properly while meeting standards such as a Windows program having "File" as the first menu item. Functional testing may be performed by someone familiar with particular standards."

"As shown in the table, there are other requirements for a software system. These include the non-functional requirements (often called the 'ilities') such as scalability, reliability, security, and performance (Footnote: Some people separate performance since it does not end in ility (or security for that matter). Some of the tests for these requirements can be expressed in questions with yes or no answers. For example, 'if there are 100 users on the system and they are all placing orders at the same time, does the system respond to each one of them in less than five seconds?' However, for other 'ilities', the question can be asked, but the answer is unknowable, such as 'Is the system secure from all threats?' The tests for many of these non-functional requirements can be performed by tools, such as load testing tools. ."

"*Usability tests* are in a separate category. One might create some objective tests, such as 'given a certain level of user, can he pay for an order in less than 30 seconds?' or 'given 100 users ranking the system usability on a scale from 1 to 10, is the average greater than 8?' But often, usability is more subjective – 'does this screen feel right to me?' or 'is this workflow match the way I do things?' Usability testing is strictly manual. There is no robot program that can measure the usability of a system. Often the customer is of the mind, 'I'm not sure what I'd like, but I'll know it when I see it'. It's difficult to write a test for that." [Constatine01]

"*Exploratory tests* are non-scripted tests [Pettichord01]. According to Cem Kaner, an exploratory tester does parallel test design, execution, result interpretation and learning. A tester like me sometimes find stuff that you wouldn't dream could be there."

Tom continues, "The term has also been applied to where all team members – you, me, and Debbie -- take on the persona of a user and go through the system based on the needs and abilities of that user. Since a system has to be working to be explored, these tests cannot be created up front. But they can be performed whenever the program is in a working condition."

"The two other testing categories are *component/integration tests* and *unit tests*. These tests relate to the architect and developer. The architect creates the overall structure of an application and designs the interfaces between the components that make up the application. He develops the tests that ensure that each component performs its responsibilities. Sam's system is small enough that Debbie takes on both the role of architect and developer. On some bigger projects, she's worked together with an architect to design the bigger modules of an application.

Unit tests are created by Debbie and other developers in conjunction with writing code. They aid in creating a design that is testable, a measure of high technical quality. The unit tests also serve as documentation for how the internal code works. Component and integration tests are the domain of the developers and architect. They verify that the modules work together to perform the desired operations. As we will see later, many of these tests are derived from the acceptance tests." [Wiki03]

"All types of testing are important to ensure delivery of a quality product. We'll be discussing mainly acceptance tests – the business facing functional tests, since these are the ones that you help create, Cathy. The tests involve collaboration between the business, the developer, and the tester. "

Where and When

Tom keeps on, "The tests can be run on multiple platforms at multiple times. Now this is a small project, so we don't have all of these platforms in the diagram. But we still run through all tests. Which platforms the tests run on depends on the environment that a test needs to run, as shown in Figure 3.2. Debbie runs unit tests on her machine. She can also run many acceptance tests, particularly if they don't

require external resources. For example, the percentage discount test could run on her machine. In some instances, she may create test doubles for external resources, but I'm getting ahead of myself. The topic of tests double will be covered in a later chapter."

*** This diagram should be in two columns - platforms on one side, tests on the other ***

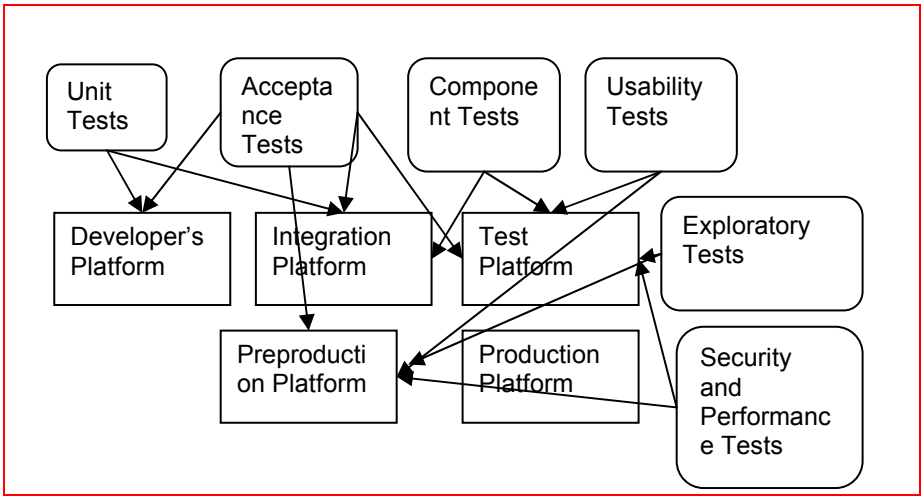


Figure 3.2 *Where Tests Run*

"On a larger project, Debbie and the other developers would merge their code on an integration platform. The unit tests of all the developers would be run on this platform to make sure that the changes one developer makes in his or her code would not affect the changes that other developers make. The acceptance tests would be run on this platform, if the external resources or their test doubles are available. In this project, Debbie's machine acts as both the developer platform and the integration platform, since she has no other developers on the project. Component tests can be run on the integration machine."

"Once all tests pass on the integration platform, the application is deployed to the test platform. On this platform, the full external resources are available, such as a working database. We can run all types of tests here. But often the unit tests are not run, particularly if the application is deployed as a whole and not rebuilt for the test platform. As a tester, I'm usually in charge of the test platform, but that doesn't mean that a developer can't also have access to it."

"Right on", interjects Debbie.

"As the customer, you get to use the user interface to see how well it works. I can try out some exploratory testing. If this were a system that required it, the security testers and the performance testers could have their first go at the application."

"The test platform may be for a single application running at a time. If the production environment had many interacting applications, we could have a second test platform that had all the applications running on it. It could be harder to run some acceptance tests, since the applications might step over each other. For example, one application might delete some data that another application was depending upon. We'll talk about how to handle that much later in Chapter 20 on Setup since there's no need to bring it up for the features we're dealing with now."

"The pre-production platform is an exact as possible duplicate of the production platform. Sometimes applications fail because there are slight differences in how environments work. You may have discovered this when you tried running some applications on a new version of your personal computer operating system."

"I'll say!" Debbie exclaims. "It took a lot of effort to get those applications working. The application vendor and the operating system vendor kept pointing fingers at each other."

Tom continues, "There are a lot of resources in a large scale system. If the versions of the resources cause an application to fail, we want to know about it before we deploy it to production. It can be something as simple as using version 6.9.1.2 of a database instead of version 6.9.1.3."

"Also, there are often some conditions that all of our analysis and tests did not uncover. It's always possible that in the real data, there lies an issue. So we try to discover any of these issues on this platform. Security and performance tests are run again, since they may have different results on this environment."

"Once the customer is satisfied with the outcome of the tests, we deploy the application to the production platform. There is still a possibility that bugs may show up. Users may do entirely unexpected things or there may be some configuration that the application does not work on. Debbie's and my measure of quality is not the number of bugs I find in the test or pre-production system, but in production. We call those 'escaped bugs' because they escaped discovery from all our testing and we buy each other coffee."

Tom resumes, "Depending on the length of time it takes to run the tests, we may run them at different times. Tests on Debbie's machine should run pretty quickly. We don't want her to get up and go for a coffee every time she starts the testing. When the code is transferred to the integration platform, we can run a series of longer tests, but not too long. There is a limit to the number of tests we can run in a short time, say 15 minutes. So we pick the most risky or the most relevant and run them. The set of tests is called a smoke test. The term is derived from electrical engineers who used to design and build a circuit. When they turned on the circuit, if it smoked, then they knew immediately that something was wrong [Wiki04]

"If there were longer running tests, then we run them at night, over the weekend, or for a week at a time on a dedicated testing platform as shown in Figure 3.3. If the tests are successful on one platform, you start running the longer set on the next platform. You probably can't imagine tests that last a week, but there a number of complex systems that require that long (and even longer)."

A Little MoreTesting

In March 2007, a large airline migrated seven million reservations from the Sabre reservation system to SHARES, another reservation system. About one and a half million reservations did not transfer correctly. Passengers could not check in for their flights. Kiosks in many cities stopped working. The conversion took place over a weekend. On Sunday, many passengers were stranded outside the security line, since they could not obtain a boarding pass. Many millions of dollars in revenue were lost, not to mention the customer dissatisfaction.

Possibly a little more testing might have gone a long way to have prevented the problem. [Fast01]

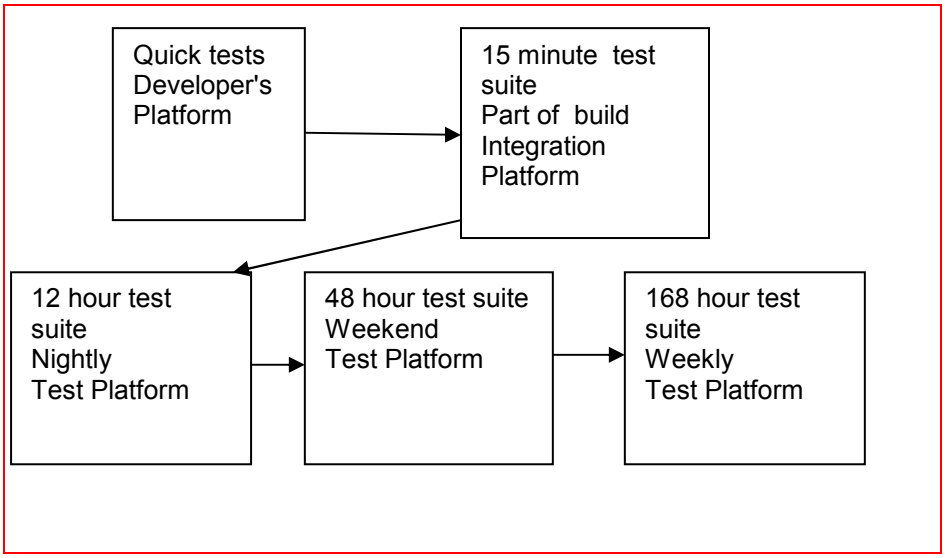


Figure 3.3 *Test Timing*

Test Types

Tom continues, "Here is another table (see Table 3.4) that shows the types of tests that we run. This shows examples of positive and negative testing. Positive tests make sure that the program works as expected. Negative testing checks to see that the program does not create any unexpected results. Acceptance tests that the customer thinks about are mostly in the 'Correct result' box. The ones that I and Debbie come up with are in the other three boxes."

Table 3.4 *Positive and Negative Testing*

	Expected Result	*Unexpected Results*
Valid Input	Correct result	Unspecified side effects
Invalid Input	Error	Any side effects

Tom resumes, "Tests are often run from the external view of the system. Bret Pettichord talks about control points and observation points. [Pettichord01]. A control point is the part of the system where the tester inputs values or commands to the system. The observation point is where the system response is checked to see that it is performing properly. Often the control point is the user interface and the output is observed on the user interface, a printed report, or a connection to an external platform. As we will see in the next chapter, it is often easier to run many tests by having control and observation points within the system itself. "

Requirements and Tests

Tom continues, "Cathy, requirements and tests are linked together. You can't have one without the other. They are like Abbott and Costello, Calvin and Hobbes, Click and Clack, Bullwinkle and Rocky, bow and arrow, cut and paste, adenine and thymine, strange and charm quarks, nuts and bolts, or your favorite duo. The tests clarify and amplify the requirements. A test that fails shows that the system does not properly implement a requirement. A test that passes is a specification of how the system works. Any test created

after the code is written is a new requirement or a new detail on an existing requirement." (**Footnote – Thanks to Scott Bain and Amir Kolsky for the discussion where this idea occurred**)

"If you come across a new detail after Debbie has implemented the code, then we need to create a new test for that detail. For example, suppose there is an input field that doesn't have any limits on what can go in it. Then you realize that there needs to be a limit. We would create tests to ensure that the limit is checked on input. The requirement that there is a limit and the tests that ensure it is checked are linked together. Since the tests did not exist before Debbie finished the code, then it is not a developer bug that the limit was not checked. It is just a new requirement that needs to be implemented. Some people might call it an 'analysis bug' or a missed requirement. But we simply say, 'You can't think of everything' and simply call it a new requirement."

Summary

- Testing areas include:
- Acceptance tests that are business facing functionality tests
- Architectural tests that check component and module functionality
- Unit tests which developer use to check functionality
- Usability, exploratory, and "illity" (reliability, scalablity, performance)
- Functionality tests should be run frequently on developer, integration, and test platforms

Chapter 4

An Introductory Acceptance Test

“"If you don't know where you're going, you will wind up somewhere else."

Yogi Berra

The Triad – Tom, Debbie, and Cathy – are in their second meeting together. Debbie describes an example of an acceptance test and four ways that an acceptance test can be executed.

An Example Business Rule

Debbie, Tom, and Cathy continue their discussion of Acceptance Test-Driven Development. Debbie is talking about a previous project where she and Tom created tests in collaboration with the customer.

“The business representative, Betty, presented us with a business rule for giving discounts that she had obtained from one of the stakeholders. The stakeholder wanted to give discounts to the firm’s customers. The discount was to vary based on the type of customer. We had already completed implementing a previous requirement that determined the type of customer. Here’s the rule that Betty gave to us:”

***Production Note - indentation in next paragraph is non-standard

If Customer Type is Good and the Item Total is less than or equal $10.00,
    Then do not give a discount,
    Otherwise give a 1% discount.
If Customer Type is Excellent,
    Then give a discount of 1% for any order.
If the Item Total is greater than $50.00,
    Then give a discount of 5%.

“This rule seems pretty clear. It uses consistent terms such as ‘Customer Type’ and ‘Item Total’. We had previously gotten from Betty the definitions of those terms. (** Ref to Ubiquitous Language and Domain Driven Design**) For example, the ‘Item Total’ did not include any taxes or shipping. But even with that consistency, we had an issue. Tom and I looked at the rule and tried to figure out what the discount percentage should be if a customer who is good had an order total greater than $50.00. Then the three of us made up a table of examples”.

Discount		
Item Total	**Customer Rating**	**Discount percentage?**
10.00	Good	0
10.01	Good	1
50.01	Good	1 ??
.01	Excellent	1

50.00	Excellent	1
50.01	Excellent	5

Debbie continues, "The first two rows show that the limit between giving a Good customer a discount or a 1% discount is $10.00. The 'less than or equal' in the business rule is pretty clear. But we wanted a test to ensure that my implementation produced that result. We put a '??' after the '1' in the third example, since it was unclear to us whether that was the right value."

"The fourth example indicates that we understand that the discount for an Excellent customer starts at the smallest possible Item Total. The fifth and sixth entries show that the discount increases just after the $50.00 point."

"Betty took this table back to the stakeholder. He looked it over and said that the interpretation was correct. He did not want to give a 5% discount to Good customers. So we removed the '??' from that result. We now had a set of tests that we could apply to the system. The correct discount amount tests is not just a single case, but includes cases for all possible combinations"

Tom interjects, "But these were not all the tests. Being trained as a tester, I like to consider other possibilities. For example, what if the Item Total was less than $0.00? I asked Betty whether this would ever happen. She said it might be possible, since the Item Total could include a rebate coupon that was greater than the total of the items. So I added the following possibilities:"

Discount		
Item Total	**Customer Rating**	**Discount percentage?**
-.01	Good	0
-.01	Excellent	1 ??

Tom explains, "It didn't seem right to apply a discount percentage that would actually increase the amount that was charged to the customer. Based on this example, Betty went back to the stakeholder and confirmed that the percentage should be 0% if the Item Total is less than 0 for any customer."

"These examples were the acceptance tests for the system. If Debbie implemented these correctly, Betty would be satisfied." Tom continued, "Now it was a matter of how we were going to use these tests to test the system."

Implementing the Acceptance Tests

Debbie states, "Tom and I needed to apply these tests to my implementation. There are at least four ways we could do this. First, Tom could create a test script that operates manually at the user interface level. Second, I could create a test user interface that allows me or Tom to check the appropriate discount percentages. Third, I could perform the tests using a unit testing framework. A standard unit testing framework is generically called XUnit. Fourth, Tom and I could implement the tests with an acceptance test framework. Let me show you some examples of how we could have dealt with each of these ways".

Test Script

Debbie continues, "In this case, the program has a user interface that allows a customer to enter an order. The user interface flow is much like Amazon or other order sites. The user enters an order and a summary screen appears, such as this one:"

(**Note: Make into a screen look-alike ***)

Order Summary			
Count	Item	Item Price	Total
10	Little Widget	$.10	$1.00
1	Big Widget	$9.00	$9.00
		Item Total	$10.00
		Discount	$0.00
		Taxes	$.55
		Shipping	$2.00
		Order Total	$12.55

"Now what Tom would have to do is create a script that either he or I would follow in order to test each of the six examples. He might start by computing what the actual discount amount should be for each case. Unless the order summary screen shows this percentage, this is the only output he can check to make sure the calculation is correct. So here's the addition to the table:"

Discount				
Item Total	**Customer Rating**	**Discount percentage?**	**Discount Amount?**	**Notes**
10.00	Good	0	0.00	
10.01	Good	1	0.10	Discount rounded down
50.01	Good	1	0.50	Discount rounded down
.01	Excellent	1	0.00	Discount rounded down
50.00	Excellent	1	0.50	
50.01	Excellent	5	2.50	Discount rounded down

"The script would go something like this:

1. Log on as a Customer who has the Rating listed in the table.
2. Start an order and put items in it until the total is the specified amount in the Item Total column on the test.
3. Check that the discount on the Order Summary screen matches Discount Amount in the table."

"Then the test would be repeated five more times for all six cases. Either he or I would do this once I've completed implementing the discount feature."

Tom interrupted, "Of course, I'd want Debbie to do this before she turned the program over to me. Better for her to get carpal tunnel syndrome, then for me."

Debbie glanced over at Tom with a wry smile on her face. She continued, "Actually, neither of us wants to get carpal tunnel syndrome. It's bad for our tennis game."

"This test that checks for the correct discount amount needs to be run, but not for all the possible combinations. You can imagine what might have happened if there were ten discount percentages for each of ten different customer types. I'd definitely have let Tom do the tests so his hand would not be in shape for our next tennis match. So let's look at the next possible way to run these tests."

Testing User Interface

"To simplify executing the tests, I could set up a user interface that connected to the discount calculation module in my code. This interface would only be used during testing. But having it would cut down on the work involved in showing that the percentage was correctly determined. The user interface might look like:"

(**Note: Make into a UI like display **)

Discount Percentage Test	
Customer Type	Good
Item Total	10.01
Percentage	1 %

"With this user interface, Tom or I could more quickly enter in all the combinations that are shown in the test table. It would cut down on the possibility of repetitive motion injuries. Our workman's compensation insurance likes that idea."

"We would still need to run Tom's original script for a couple of instances to make sure that it was properly connected up to the discount percentage module. But unless there was a large risk factor involved, you might just run it for a few cases such as:

Discount			
Item Total	**Customer Rating**	**Discount percentage?**	**Discount Amount?**
10.01	Good	1	0.10
50.01	Excellent	5	2.50

"This user interface has penetrated into the system. It exposes a test point within the system that allows easier testing. Let me give you an analogy showing the differences between this method and Tom's original test script. Suppose you want to build a car that accelerates quickly. You know you need an engine that can increase its speed rapidly. If you could only check the engine operation as part of the car, you would need to put the engine in the car and then take the car on a test drive. If you had a test point for the engine speed inside the car, you could check how fast it sped up without driving the car. You could measure it in the garage. You'd save a lot of time in on-the-road testing if the engine wasn't working properly. That doesn't mean you don't need to test the engine on the road. But if the engine isn't working by itself, you don't run the road test until the engine passes its own tests. "

"If you're not into cars, let me give a context diagram, as in Figure 4.1. The Order Summary Screen connects to the system through the standard user interface layer. The Discount Percentage Screen connects to some module inside the system. Let's call that module the Discount Calculator. By having a connection to the inside, a tester can check whether the internal behavior by itself is correct.

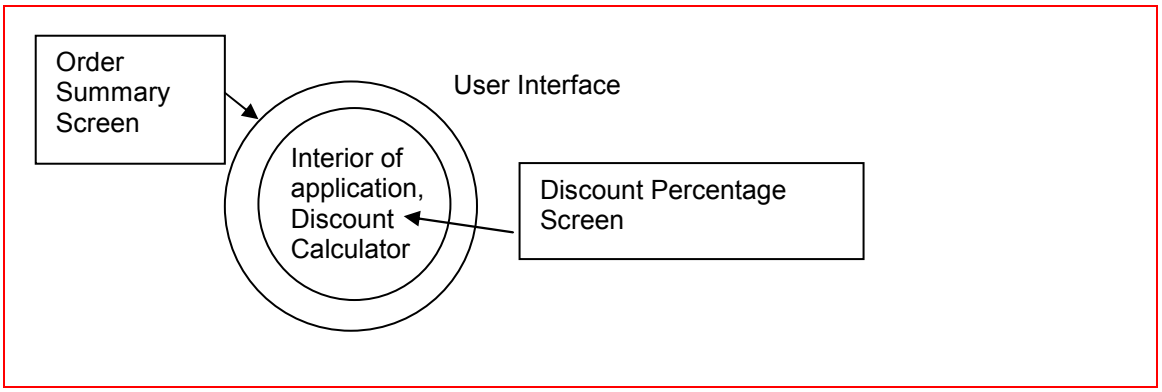


Figure 4.1*Context Diagram*

XUnit Testing

"The next way I could perform the testing is to write unit tests for the discount calculator. These unit tests are usually written in the program language. Here's a sample of what these tests look like in Junit The test would look similar in TestNG, but the order of the parameters would be reversed. I know you're not a programmer. But as you can see, it can be a little hard to see exactly what is being tested."

```
class DiscountCalculatorTest
{
@Test
testDiscountPercentageForCustomer()
   {
   DiscountCalculator dc = new DiscountCalculator()
   assertEquals(0, dc.computeDiscountPercentage(10.0,
     Customer.Good));
   assertEquals(1, dc.computeDiscountPercentage (10.01,
     Customer.Good));
   assertEquals(1, dc.computeDiscountPercentage (50.01,
     Customer.Good));
   assertEquals(1, dc.computeDiscountPercentage(.01,
     Customer.Excellent));
   assertEquals(1, dc.computeDiscountPercentage(50.0,
     Customer.Excellent));
   assertEquals(5, dc.computeDiscountPercentage(50.01,
     Customer.Excellent));
   }
}
```

"Luckily Tom has done some programming, so he can read this code. However any time we make a change in the tests that Betty and the stakeholder can read, I also have to make a change in these tests. That's a little bit of waste. And to effectively produce software, we'd like to eliminate that waste if possible. So Betty, Tom, and I selected the next mode of testing."

Automated Acceptance Testing

"The three of us agreed that the examples in the table accurately reflected the requirements and there would be less waste if the table did not have to be converted into another form for testing. There are several available test frameworks that can use this table or a slightly altered form of this table to directly drive the tests. I'm not going to get into the details of them at this time. They are in Appendix 2."

"What I would like to show is how we use one of these frameworks. The one we use in this example is the Framework for Integrated Tests or known as Fit. It was developed by Ward Cunningham [3][Cunningham01], [Cunningham02] and turned into Fitnesse by Bob Martin [Martin01]."

"With Fit, you describe the tests with a table similar to the one we used with Betty. I'm not going to get into the syntactic details or otherwise it might confuse you at this point. I'll use the same table that we started with. And besides, we're not trying to push a particular test framework. So getting into the minute details is beyond the scope of this book."

"Here's the test. It looks just like the table that Betty presented to the stakeholder."

Discount		
Item Total	Customer Rating	Discount percentage?
10.00	Good	0
10.01	Good	1
50.01	Good	1
.01	Excellent	1
50.00	Excellent	1
50.01	Excellent	5

"Now when we run this table as a test, Fit executes code that connects up to the Discount Calculator. It gives the Discount Calculator the values in Item Total and Customer Rating. The Discount Calculator returns the Discount Percentage. Fit compares the returned value to the value in the table. If it agrees, the column shows up in green. If it does not, it shows up as red. You can't see the colors in black and white. So light grey represents green and dark grey represents red. The first time I ran the test, I got the following table as the output of Fit. "

(** Note: Make the table have formatting that looks different for these rows".

Discount		
Item Total	Customer Rating	Discount percentage?
10.00	Good	0
10.01	Good	1
50.01	Good	Expected 1 Actual 5
.01	Excellent	1
50.00	Excellent	1
50.01	Excellent	5

Tom winks and says, "With this table as the results, it was apparent there was an Discount Calculator. Once it was fixed, Betty saw the passing tests as confirmation that the calculation was working as desired."

[3] Ward's latest project is creating the Swim system for Eclipse which applies TDD to business processes [Cunningham03]

Summary

- Examples of requirements clarify the requirements
- The examples can be used as tests for the implementation of the requirements
- Tests for business rules can be executed in many ways
  - Creation through user interface of transaction that invokes business rule
  - Development of a user interface that invokes the business rule directly
  - A unit test implemented in a language's unit testing framework
  - A automated test that communicates with the business rule module

Chapter 5

The Overall Project

"When you are inspired by some great purpose, some extraordinary project, all your thoughts break their bonds; your mind transcends limitations, your consciousness expands in every direction, and you find yourself in a new, great and wonderful world."

Patanjali

The team meets for a chartering session to develop the goals and objectives for Sam's idea for a new system. The charter is the first step in the project (see Figure 5.1). Then the team holds the initial requirements elicitation workshop that creates high level requirements and high level acceptance criteria.

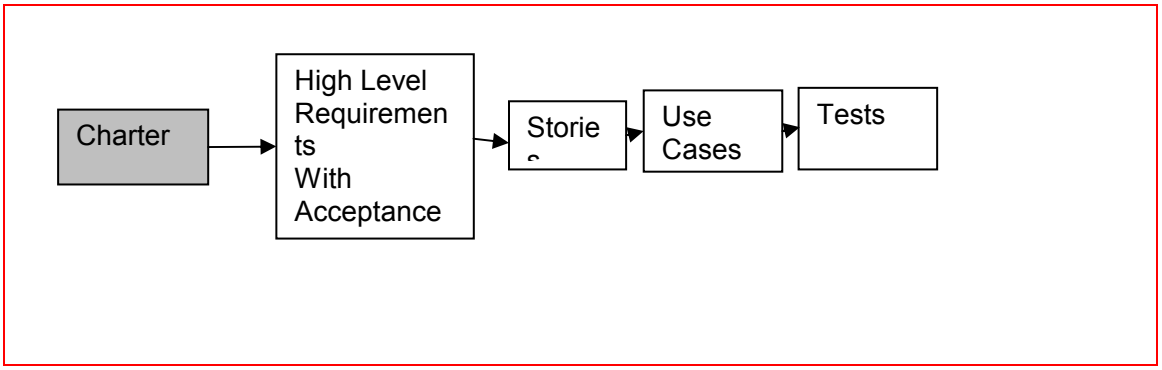


Figure 5.1 *The Project Steps Start with the Charter*

The Charter

The story continues. Sam rents a conference room for the initial project meeting. There is quite a crowd. Sam is the sponsor of the project. Cathy is the business customer. Sam's sister Mary and her husband Harry who often work in the store, and Cary, who clerks in the store are there as potential users. And of course Debbie and Tom, the development and test units, are participating as shown in Figure 5.2.

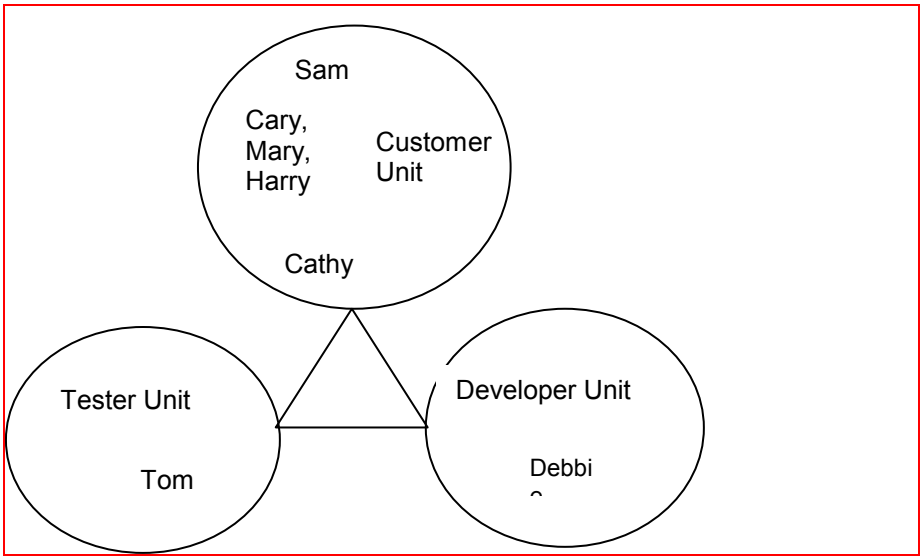


Figure 5.2 *The Team*

The purpose of the meeting is to create a project charter. The charter explains the purpose of the project, its goals, and the objectives it should meet. This information will be posted on the project tracking board to keep everyone focused. The meeting involves more than just Sam and Cathy so that everyone can get an idea of the big picture, understand the issues involved, and agree that the project's objectives are feasible.

A basic charter has three parts to it. First is the vision or purpose which describes how the project fits into the overall enterprise vision and purpose. Second is the goal of the project, which is a subjective measure of the success of the project. A goal is a brief statement of intent. You may have more than one goal for a project, but having too many goals can dilute the focus. Third are the objectives, which are specific measures for the success of a project. Objectives should be SMART. SMART is an acronym that has many interpretations [Project01]. In the version here, SMART means:

- Specific – exactly what is going to be done or a specific outcome?
- Measurable – can the outcome be measured?
- Achievable - Can it reasonably be accomplished?
- Relevant – does it support a goal?
- Time-boxed – when will the objective be achieved?

Sam presents his vision for the project. The store needs a CD rental information system. His initial goal is to reduce the staff time to process rentals. Everyone agrees that is primary. Cary suggests that the system could offer more services to customers, such as informing if a CD is overdue or that a CD they want has just been returned. All agree that sounds like a good additional goal.

The discussion starts revolving around objectives. Without detailing who-says-what-when, the customer unit (Sam, Cathy, Mary, Harry, and Cary) usually proposes objectives. If Debbie and Tom (the developer / tester units) think that an objective is not achievable within the time box, then the objective misses the 'A' part of SMART. The measure or the time box has to be change so that the objective is agreed upon or the objective has to be dropped. The meeting ends with everyone agreeing on some SMART objectives, listed here in the charter:

Project Charter

Vision:

- CD Rental information system

Goals:

- Reduce staff time to process rentals
- Offer more services to customers

Objectives:

- Within two months, CD check-outs and returns will be processed in 50% less time.
- Within three months, at least one new service will be offered to customers.

The objectives are the acceptance tests for the overall project. The first objective requires that the current amount of time that it takes to process a CD be measured. In two months, the amount of time will be measured again. If the second measure is not 50% less than the first measurement, the test has failed and the objective has not been met. Did the project still produce business value by reducing the time by say 49%? Yes. But it did not deliver the full business value that was the reason the project was approved.

A Simple Charter

A $200 million dollar project had a very simple charter. It was defined by the following objective:

- 70 mpg

This was the charter for the Honda Insight, a new hybrid vehicle. The single focus of the charter drove the engineering design. When engineers needed to make decision, they consulted the objective. Cutting weight is a large factor in increasing mileage. You might imagine the discussions:

"Seatbelts? Well, they weigh three pounds. Delete them." There were implicit constraints. The car had to be at least sellable.

"Air conditioning? It takes two mpg. Skip it." The two miles per gallon would count against the mileage if air conditioning was factory installed. So the dealers had to do install it.

The total capacity was to 365 pounds. That is the total for two people and cargo.

The goal was not to sell a lot of cars, but make the mileage objective.

And they did it.

P.S. Do you know what the charter for your project is?

The term "one new customer service" in the second objective may seem a little vague. If necessary, the group could write down separately what is meant by a customer service. But since everyone is at the meeting, a general consensus is reached. Detailing which is the most important customer service to offer is deferred to a requirements elicitation meeting.

Objectives Should Be Measurable

A project at one company had a purpose of making their website more user-friendly so that people would stay there longer, thus increasing the possibility that the people would buy more products and drive revenue. They started with an objective that read something like:

- Within six months, the site should be more user-friendly.

The issue with this as an objective is that it is almost subjective. It needs to be stated in measureable terms. The measurement must be relevant to the goal. There are many ways to perform measurements such as measure the time it takes for a user to perform certain operations. Detailed measurements can help determine why a particular program is or is not user-friendly. But for purposes of a simple high level measurement, one might use a survey such as the System Usability Scale (a measure of 0 to 100). [Usability01] . Now the objective can reference a measureable result, such as:

- Within six months, the web site shall have an increase of 10 points on the System Usability Scale.

The High Level Requirements

A business must have capabilities in order to run. In Sam's case, the capabilities include keeping track of CDs and collecting payment for rentals. The capabilities today are implemented through manual features - a set of paper index cards and a cash register. The new system will have different features that provide these capabilities. In addition, the charter states that it should have at least one new capability for providing a customer service.

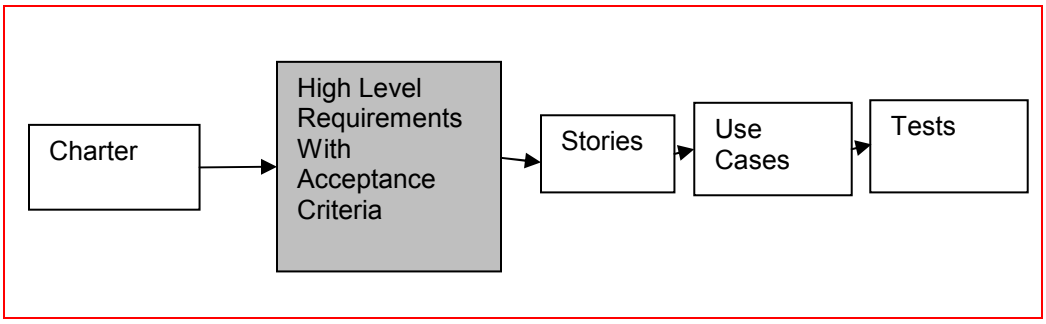


Figure 5.3 *After Charter, Elicit Requirements*

The entire team meets again for a requirements elicitation workshop, the next step as shwon in Figure 5.3 Briefly, here is what goes on. Tom, who has had some facilitation training, manages the brainstorming workshop [Gottesdiener01], [Gottesdiener02], [Gottesdiener03], It is also natural for him to facilitate as he does not have any investment in the outcome. For a few minutes, he has all the participants individually write down ideas for features, one per index card. Tom suggests that everyone should not hold back on their thoughts. They should write down any ideas that they can imagine which may pertain to the charter. The limitation comes later when the ideas are matched to the objectives of the charter.

Everyone then places their cards on the table and they match up ideas that are almost the same. If there are natural groupings of ideas, they place the cards close to each other. The ideas that were grouped together show up underneath the overall idea. The results are as follows:

Check out and check in

- Bar code reader for recording CDs and Customers cards with Customer ID
- In-counter bar code reader for faster scanning

Reservation system for CDs
Credit card charging, to eliminate cash
Discounts for frequent renters

- Every so often, give a free CD rental

A CD catalog of available CDs

- CD catalog of all CDs so renters could select one to rent

For multiple stores, way to return CD to any store
For multiple stores, way to determine which stores have a CD
Hookup with a video rental store to offer combined rental discounts
Have a party for customers who rent lots of CDs that month

The ideas need to be whittled down a bit. The customer unit needs to decide which features should be part of the scope of the project and which should be implemented first. A quick simple way to cut down and order a list is dot voting. In dot voting, each participant gets a certain number of votes and marks his votes with a dot on the corresponding card. The number of votes per participant is the total number of items divided by three. In this case there are nine items, so each voter gets three votes.

To make the voting more in-line with the level of involvement of the participants, a set of participants may be grouped together. If Mary, Harry, and Cary each had the same number of votes as Sam and Cathy, then part-time participants could outvote full-time participants. So a total of one vote each is given to Mary, Harry, and Cary; three votes to Sam; and three votes to Cathy. The votes are on the highest level items, such as check-out and check-in, not on the details that are grouped with them. The items that received votes are:

Check out and check in (3)
Credit card charging, to eliminate cash (2)
Reservation system for CDs (2)
Discounts for frequent renters (2)

The remaining ideas are not tossed away. They are captured for revisiting once the selected features are implemented. Now that the features are selected, the entire team examines them to see what the high level acceptance criteria are for each of them. As a tester, Tom has valuable input into this process. If the team cannot come up with a high level idea of how to check that a feature is implemented correctly, the feature needs more definition. Working together, the team comes up with the following:

Check-out and check-in

- Check out a CD, make sure the details are correct and it's recorded as rented on the inventory
- Check in a CD, make sure that any late rental fees are computed, and it's recorded as returned.

Credit card charging, to eliminate cash

- Check out a CD and see if a charge is recorded
- Check in a CD and see if late rental fees are charged

Reservation system for CDs

- Reserve a CD and see if reserver is notified when CD is returned and available

Discounts for frequent renters

- Rent several CDs and see if the rental charges are reduced or free on a subsequent one

Now that the features are agreed upon, the next step is to start developing some. Sam and Cathy agree that the first two on the list should be the first ones to develop.

Summary

- A charter states a project's vision, goals, and objectives
- Objectives should be specific, measurable, agreed upon, relevant, and time-boxed
- Objectives represent acceptance tests for the whole project
- High level requirements should support the objectives

- High level requirements have high level acceptance criteria

Chapter 6

A User Story

"It is better to take many small steps in the right direction than to make a great leap forward only to stumble backward."

Chinese Proverb

The triad meets to develop stories from features. Debbie explains roles, their attributes, and personas. She introduces a story card template. Tom shows how acceptance criteria can determine story size. He lists the INVEST criteria for stories.

Stories

The features the team agrees upon need to be broken down into smaller pieces, the next step shown in Figure 6.1. It is easier to devise acceptance tests for smaller pieces than for an entire feature. The tests may be based on some common setup, but each test specifies only one behavior for that setup.

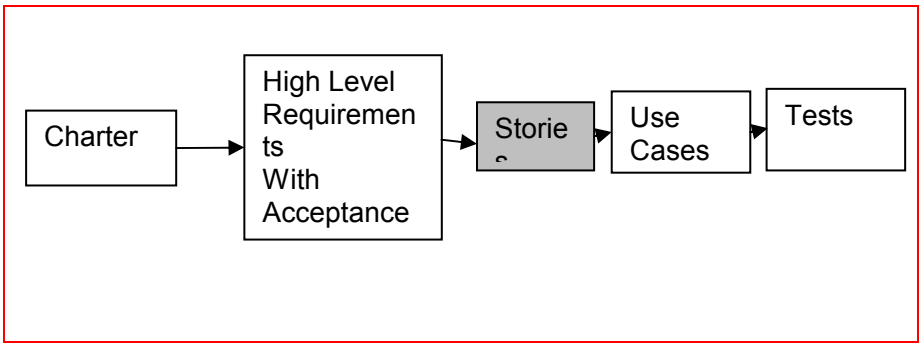


Figure 6.1 *After High Level Requirement, Create Stories*

The team gets together again, this time without Sam. As sponsor, he has agreed to and is happy with the features that are going to be implemented. He will come back as they are developed to see how things are going. Cathy will develop the stories, getting suggestions from Cary, Mary and Harry as she goes along. Cathy has the necessary business and domain knowledge to create the stories. Debbie and Tom work with Cathy to help her break down the stories and to get a first hand understanding of what the stories are all about.

In the first story workshop, they are going to break down the first few features into stories. Debbie begins, “A requirement story is a small portion of a feature. Many of the requirement stories are called user stories because they involve something that a user wants to do with the system. Sometime a

requirement story is just a constraint on the system, such as using open-source software to avoid license fees or writing the program in Java since that's the corporate standard. Those are often called constraint stories. The acceptance criteria for a constraint story are pretty well specified by the constraint itself, so we won't deal with those further."

Debbie explains, "Let's start to break down the features into stories. The features we are going to start with are:

Check out and check in CDs
Credit card charging, to eliminate cash

"Tom and I use a story format that comes from Extreme Programming (**Ref**). The form is:"

As a <role>, I want to <do something> so that <reason>

"The role represents the user, the 'do something' is what the user is trying to accomplish, and the 'reason' is why they are doing it. The 'role' and 'do something' are the critical parts. The 'reason' is often helpful, but not required. An example of this form is:"

As the clerk, I want to check out a CD for a customer so that I can keep track of who has rented it.

Roles

"Before we start on the user stories, we need to come up with some roles that are going to be involved in the stories. These roles are not necessarily specific people, but the 'hats' that people wear in the rental process. These roles are important. When we gather the details for a story, we want to ensure that someone who plays that role is available to provide them. When Tom does usability testing or exploratory testing, he will take on a role to see how the system functions from that perspective."

"We use the same method that we did in coming up with features. Everybody brainstorms by themselves and writes down potential roles. We then put the cards on the table, match them up, and group them."

After thinking, writing, and grouping, the team came up with the following roles. Each role is clarified by listing its responsibilities.

Clerk – checks out CD and checks them in
Inventory Maintainer – responsible for keeping track of overall CD inventory
Finance Manager – responsible for all monetary transactions such a rental payments and late rental fees
Renter – one who pays for CDs with cash or in the future with a credit card

Tom notes, "Cathy plays two of these roles – Inventory Maintainer and Finance Manager. The roles are separate, since they have different interests and points of view."

"Now that we have the roles, it's useful to come up with attributes for the roles. These attributes give me a better idea of how to design the system and Tom how to test the system. For each role, you determine:"

- Frequency of use – how often they use the system
- Domain expertise – in the area that the system is designed for
- Computer expertise – in using a computer
- General goals – in what is desired, such as convenience, speed, etc.

"Let's take the role of clerk. Is there anyone else other than Cary who works as a clerk and what are their backgrounds?" Debbie asks.

Cathy replies, "Harry and Mary. Cary works there every day and is a computer whiz. His dad Harry is a retired English professor who fills in every now and then. He gets all his information from the books in the stacks at the library rather than from the Internet. Mary still works as a French professor. Her computer skills are probably more in line with Harry's than with Cary's".

Debbie goes on, "To avoid giving them names loaded with connotations, let's call the two types Full Time and Part Time Clerks." The team develops the following sets of attributes:

Full Time Clerk

- Frequency of use – every day
- Domain expertise – excellent
- Computer expertise – excellent
- General goals – speed, as few keystrokes as possible

Part Time Clerk

- Frequency of use – one day a week
- Domain expertise – understands the general area
- Computer expertise – low
- General goals – foolproof, lots of helpful reminders

Debbie continues, "We could create a persona for each of these sets of attributes. A persona is an imaginary person described with lots of details. It helps me and Tom to envision a actual user, rather than just a dull set of attributes. I don't think we need one for the 'Full Time Clerk', so let's do one for the 'Part Time Clerk'. We'll use a different name for the persona."

Here's what they came up with:

"Larry comes in one day a week to help check in and check out CDs. He wants to be able to do that without making too many mistakes. He's familiar with a regular typewriter keyboard, but doesn't understand special keys like Ctrl and Alt."

Debbie comments, "This persona gives me a good picture to keep in mind for whom I'll be developing the user interface. Now let's start on the stories themselves. "

Stories for Roles

Over the next few hours, some additional stories that develop are:

- As the Clerk, I want to check out a CD for a customer.
- As the Clerk, I want to check in a CD for a customer.
- As the Inventory Maintainer, I want to know where every CD is – in the store or rented.
- As the Finance Manager, I want to know how many CDs are turned in late and the late charges that apply
- As the Finance Manager, I want to know how much is being charged every day so that I can check the charges against bank deposits.

Debbie introduces the story card template. "Here's a template (Figure 6.2) that we've used on past jobs to record the stories. By keeping the information on a single card, you can easily re-order them. And as I develop, I concentrate on a single story, although I may consider the big picture in which the story fits."

(** Note: Maybe make below into a card-like image **)

Story ID		**Name**		**Related Stories**		
Description	.					
Acceptance Criteria						
Details		**Business Value Estimate**		**Story Point Estimate**	**Bang For The Buck**	
Notes						

Figure 6.2 *A Story Card Template*

"We put an ID on each story so that it's easier to reference by other stories. The purpose of the 'Related Stories' is to indicate if this story has other stories that may need to be considered together. 'Name' is just a shorthand description so we don't have to remember what 'Story 42' is all about. 'Acceptance Criteria' are like the ones that we developed for the features. Since a story is more specific, the tests are more specific. We will get into the actual tests when we begin work on the story."

"The 'Details' give the name of the person who knows more details about the story. That person could be someone who fills the role or who is a subject matter expert (SME). 'Notes' are any other things that are captured about the story. 'Business Value Estimate' is the customer's unit relative estimate of this story's worth compared to other stories. 'Story Point Estimates' is the developer's unit estimate of effort to implement the story relative to other stories. 'Bang for the Buck' is a rough return on investment relative to other stories. It is computed by dividing the Business Value by the Story Points."

"We could estimate Business Value and Story Points for this project. But since we're concentrating on acceptance tests, that estimation will be shown in Appendix 4.. We would track the Business Value of stories that are completed, so you could get an idea of the project's progress. You could use Bang for the

Buck to determine whether a particular story should even be implemented. I do need to note that the Story Points estimate includes both implementing the code and testing it. So Tom and I jointly make that estimate. ”

“Based on our discussion, Figure 6.3 shows what the story card for the first story might look like. I’ve filled in values for Business Value, Story Points, and Bang for the Buck, so that we have a baseline for estimating other stories relative to this one, if we chose to do so.”

Story ID:	1	**Name:**	Check-out CD	**Related Stories:**	None
Description:	As the clerk, I want to check out a CD for a customer.				
Acceptance Criteria:	Check out CD. See if recorded as rented and rental info is correct.				
Details:	Cary	**Business Value Estimate:**	100	**Story Point Estimate:**	13	**Bang For The Buck:**	8
Notes:	Make sure user interface also works for Larry, the Part-Time Clerk				

Figure 6.2 *A Filled-in Story Card*

The discussion on stories continues. More stories come up:

Check In CD - As the Clerk, I need to check in a CD.
Report Inventory - As the Inventory Maintainer, I want to see a report of where the CDs are – on rental or in the store.
Charge Rentals - As the Finance Manager, I want to submit a credit card charge every time a CD is rented so that the store does not have to handle cash

Debbie continues, “Check-in CD is related to Check-out CD, so we would show that on the story card as a related story. You can’t check in a CD unless it has been checked out. So that impacts the sequence in which the stories will be developed.”

“As we create each story, we need to list the acceptance criteria for the story. The criteria will be expanded into specific acceptance tests when we work on the story.” The team comes up with the following tests

Check In CD

- Check in CD. Check to see it is recorded as returned.
- heck in a CD that is late. Check to see that it is noted as late.

Report Inventory

- Check out a few CDs. See if report shows them as rented
- Check in a few CDs. See if report shows them as in store.

Charge Rental

- Check in a CD. See if rental charge is correct. See if credit charge matches rental charge. See if charge is made to the credit card company. Check that the bank account receives money from the charge.

Acceptance Tests Determine Size

"That last story, the 'Charge Rental' seems too big from the acceptance criteria," Tom suggested. "There are tests associated with the rental and tests associated with the credit card company. If we recognize that a story is too big at this point, it should be broken down into smaller stories. The smaller stories can be easier to develop and test. If we discover the number of specific acceptance tests is large when we detail the story, then we can break it into two stories at that point. "

"The 'Charge Rental' story feels like it could be broken into at least two stories. One might be 'Compute Rental Charge' and the other 'Process Charge'. The tests underneath each of these stories would be:"

Compute Rental Charge.

- Check in CD. See if rental charge is correct. See if credit charge matches rental charge.

Process Charge

- See if charge made to the credit card company. Check that bank account receives money from the charge.

Tom continued, "In this case, Cathy, you can come up with acceptance tests for both of these stories since they are both business related. These two stories are related back to 'Charge Rental'. If you were estimating business value, these two stories would not have any value, but 'Charge Rental' would have value. We cannot run an acceptance test for 'Charge Rental' until both of these are complete. No business value would be credited until we could show that story was completed."

"When we get to the details, Debbie and I may find we need to break up stories into smaller ones, that are technical and which you may not understand. These stories we call 'developer stories'. Debbie and I come up with them to cut down the size of the stories. If we had multiple teams, we could break up a story into ones that each team would work on. It is the responsibility of the developer unit to create acceptance tests for developer stories. (**Footnote we'll talk about this more later**)"

Customer Terms

Debbie announces, "Now we need to agree on common terminology. It seems that we are using the term 'charge' in many ways. For example, 'charge' as what you charge for a rental and a charge made on a credit card. This can be confusing later on. Cathy, we need to state the terms in business language, not computer language. So we can all agree on the terms, let's write a glossary. (** Ref again ubiquitous language**)" The team created the following:

> Rental Fee - amount due for a rental at checkout
> Late Fee – amount due if rental is late when checked in
> Card Charge – amount charged to customer's credit card for any reason.

"Once we've come up with the terms, we need to use them consistently. So let's re-write the stories to use these words," Debbie continued. "Let's take these two stories as an example."

> Compute Rental Fee.
>
> Check in CD. See if rental fee is correct. See if card charge matches rental fee.
>
> Process Card Charge
>
> See if card charge made to the credit card company. Check that bank account receives money from the card charge.

INVEST Criteria

Tom began. "The INVEST criteria for requirement stories was developed by Bill Wake (**Ref**). INVEST stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable. Debbie described some of these aspects already. I thought I'd give a short summary of the criteria so you'll understand how Debbie and I approach evaluating stories. As we create each story, we examine it against the INVEST criteria. That stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable"

"*Independent* means that each story can be completed by itself, without dependencies on other stories. Often it's the case that a sequence of stories exists, such as the check-out and check-in stories. Some people term this sequence a saga. Although there is a relationship between the stories, check-out can be completed by itself, and later the check-in story can be done. But it would be hard to do the stories in reverse – check-in first and then check-out."

"*Negotiable* means that the triad, we and you, have not made a hard and fast determination of exactly what is in the story. We will collaborate on that when we start working on the story."

"*Valuable* means that the story has a business value to you Cathy. That's the reason that Debbie and I often suggest having the customer unit put a business value on each story. If you cannot ascribe a business value, then perhaps the story should not be done. Any developer story we create must be related to some story to which you have assigned a business value."

"*Estimable* implies that Debbie and I can come up with some sort of rough estimate as to how long it will take us to complete the story. If we lacked knowledge about the business domain or were implementing the story in some completely new technology, we might not be able to give an estimate. If you needed a rough estimate to justify spending money on the story, we would spend some time investigating the domain or the technology."

"*Small* stories can be completed in a single iteration or in a reasonable cycle time. If a story cannot be completed in a single iteration, then it's hard to track progress and chances are it is too big a story to easily comprehend. Preferably we break big stories (which some people call epics) into smaller stories each of which you can understand. Otherwise we may need to break the stories down for technical reasons into developer stories to facilitate coding, as Debbie mentioned before. When we broke down 'Charge Rental' into 'Compute Rental Fee' and 'Process Card Charge', we were making the stories meet this criterium."

"*Testable* means that the user can confirm that the story is done. Having acceptance tests makes it testable and passing them shows that the system meets your needs. As we will see later, having acceptance tests that can be automated makes sure that previous stories are not broken when we implement new stories"

Tom continues, "There are other reasons that you might want to break a story into multiple stories, even if the story meets these criteria. All the details of a story may not need to be completed to deliver business value. So the details not currently required might be incorporated in a new story. You might think that some aspects of a story are riskier than normal. So you might create a story to investigate those aspects early on in a project. I don't think we'll see these things occur on this project, but in some projects this must be considered."

The triad spends a few minutes reviewing each story against the INVEST criteria. Then Debbie finishes, "I think we're ready to develop more details and specific acceptance tests."

Summary

- Requirement stories can be user stories or constraint stories.
- Every user story has a role and an action and usually a reason.
- Roles are the parts people play in a process, not individuals.
- Stories should be written in the customer's language.
- Stories should meet the INVEST criteria – Independent, Negotiable, Valuable, Estimatable, Small, Testable.
- Each story should have acceptance criteria.
- Acceptance criteria can help determine the size of stories.

Chapter 7

Collaborating On Use Cases

"What we've got here is a failure to communicate"

Captain, Road Prison 36, Cool Hand Luke

Debbie explains to Cathy how to create use cases (see Figure 7.1). The triad constructs a use case for the user story about checking out a CD. Issues in collaboration are discussed.

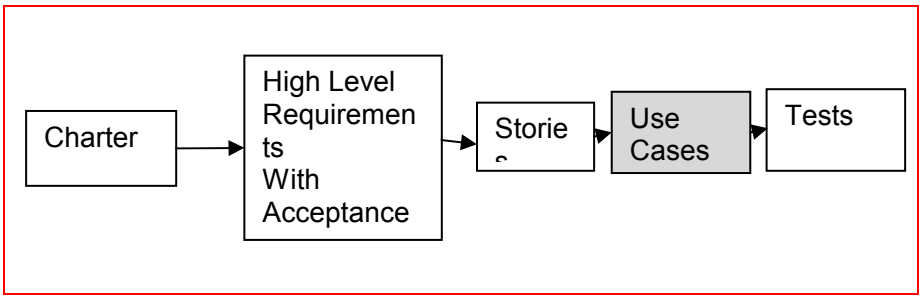


Figure 7.1 *Use Cases*

Use Cases from User Stories

Debbie explains to Cathy about how they discover the details of a user story. We use a common technique called a use case. (**Footnote – Event/Response is shown in another chapter **). "The use case describes a sequence of actions and reactions between the user and the software. There are several formal templates for a use case, but Tom and I prefer a simple one. It's adapted from Alistair Coburn's book on use cases [Cockburn02]. The first story we're going to work on is Check-out CD."

Story ID:	1		**Name:**	Check-out CD	**Related Stories:**	None	
Description:	As the clerk, I want to check out a CD for a customer.						
Acceptance Criteria:	Check out CD. See if recorded as rented and rental info is correct.						
Details:	Cary	**Business Value Estimate:**	100	**Story Point Estimate:**	13	**Bang For The Buck:**	8
Notes:	Make sure user interface also works for Larry, the Part-Time Clerk						

Debbie continues, “Often the use case is part of a workflow that either involves other use cases or actions that occur outside of the software system. Let’s track the steps which occur when one of your customers rents a CD with the manual process. Cathy, can you describe the current flow for checking out a CD?”

“Sure”, Cathy replies. She writes on a whiteboard the steps. After a few additions and corrections, the steps look like:

Customer selects a CD from the cases on the wall.

- Cases are empty, but just have cover page

Customer brings CD case to the Clerk
Clerk gets actual CD in a case from shelf behind counter
Customer presents his driver’s license
Clerk pulls out the rental card from the CD case
Clerk writes down the customer’s name and the current date on the rental card
Customer signs the rental card
Clerk files the rental card in the CD case with the title and stores the card in a box on the back shelf.

Debbie starts, “The software system will not replace all of these steps. A bigger system, like those red DVD rental kiosks might, but not the system we’re replacing. So we only need to concentrate on the steps involved with recording the rental itself. Based on your current workflow, in the new system, these might be:”

Clerk enters customer identification and CD identifier into the system
System records the information
System prints a form that the customer signs

“These steps form the main course for a use case. Some people call this the happy path, since it assumes that nothing goes wrong. The template for a basic use case looks like this:”

Name: Identifier to easily reference it by
Description: Brief note
Actor: Who initiates the use case
Pre-conditions: What must be true before use case is initiated
Post-conditions: What is true if use case successfully executes
Main Course: Steps that show sequence of interactions

“The Actor is almost always the role (the user) in the user story. The name of the use case can be the name of the user story. The brief description can be the same as the description on the user story. The pre-conditions describe the required state of the system prior to starting the main course. The post-conditions are how the state of the system has changed. They describe the tests we need to run to ensure that the implementation successfully implemented the use case. The pre-conditions represent the setup required for those tests. So based on the story and the steps, the basic use case looks like”:

Name: Checkout CD
Description: Check out a CD for a customer
Actor: Clerk
Pre-conditions: Customer has identification. CD has identity.
Post-conditions: CD recorded as rented. Rental form printed.
Main Course:

1. Clerk enters customer identification and CD identifier into the system
2. System records the information

Debbie states, "Now that we've identified the main course, we can add additional information to the use case. During the use case, conditions can occur that do not allow it to reach its post-conditions. We call these conditions exceptions. There are exceptions that can happen in almost any use case. For example, you could have a power failure and the computer could go down. Or there could be some inexplicable software failure." Tom interrupts, "But that never ever happens with Debbie's code".

After a nod to Tom, Debbie resumes, "We deal with those sorts of exceptions with an overall response scheme, such as filling out the rental contract manually. There are specific exceptions that can occur during the main course. For example, it's possible that the customer identification is not recognized when the clerk enters it. We identify this exception with an item that is numbered with the step in the main course where it could occur. We add a letter to denote it as an exception. So this might look like:"

Exceptions:

1a. Customer identification not recognized.

"We could have the clerk enter it again. So we make note of that action with the exception. We put that beneath the exception, such as:"

Exceptions:

1a. Customer identification not recognized.
Repeat step 1.

"But suppose that this step is repeated and the customer identification is still not recognized. Now this is a decision for you Cathy. It could be that the customer identification is not very readable or it could be a fake customer identification. It's not up for the developer unit to determine how to handle this exception. It's the business unit's responsibility. What should the system do?" Debbie asked.

Cathy replies, "I suppose the clerk could take down additional information from the customer and rent it anyway. We might lose a CD or two for fake ids, but we would avoid making real customers unhappy. I'll check with Sam, but for now, let's do that."

Debbie says, "Okay, so let's call those steps 'record customer id' and 'checkout manually'. You can come up later with the exact details. Let's put that down. Since it is a different exception, we give it a different letter. So the two exceptions that can occur during step 1 are:"

Exceptions:

1a. Customer identification not recognized on first try
Repeat step 1.

1b. Customer identification not recognized on second try

Clerk performs "record customer id" and "checkout manually"

Use case exits

Debbie asks, "Do you have any business rules that apply to the rental process? Our definition of a business rule is something that is true, regardless of the technology."

"We do have one that is hard to enforce, given the way we do things now," Cathy replies. "Sam and I agreed that a customer should not be able to rent more than three CDs at any time. The rule limits our losses in case the customer skips out on us. It also keeps more CDs in stock for other customers."

Debbie responds, "So you want the check-out abandoned in that case. Let's get that one down. Later on you can change your mind, such as increasing the limit for a particularly responsible customer. But that would involve a little more coding."

Exceptions:

1c. Customer violates CD Rental Limit
Clerk notifies customer of violation

Use case abandoned

Business Rule:

- CD Rental Limit
  A customer can only have three CDs rented at any one time

"One other facet of use cases is the alternative. An alternative is a flow that allows the use case to be successful even if the some condition occurs. For example, the printer might jam when printing out the rental contract. In this case, the clerk could fill out the contract manually. So we'd add an alternative to step 3:"

Alternatives

3a. Printer jams.

Clerk fills out contract.

Use case ends successfully.

"This use case is fairly straightforward. If there were several alternatives, we'd make up separate use cases to keep each one simple. We know from experience with testing that each alternative always requires more tests. If the number of tests for an alternative seemed large, then we definitely would split up the use case. If it took me a while to implement an exception or you could use the system without the exception being handled, then we'd make up separate stories for either an individual or a group of exceptions. Those stories would be related back to the one for the main use case."

The Tests

Debbie continues, "Now that we have the use case for this story, it's time to outline the tests to write against it. We need at least one test for the main course, each exception, and each alternative. Later we will make up specific examples for each of these tests. The use case suggests these tests:"

Rent a CD – main course
One Bad Customer ID – enter customer id wrong once
Two Bad Customer IDs – enter customer id wrong twice
CD Rental Limit – customer who has three CDs and rents another one
Printer Jam – simulate a printer jam (maybe out of paper)

"As I mentioned before, the pre-conditions convert to the setup for these tests and the post-condition are the expected results. If there is an exception, we should see something other than the post-condition, since the use case did not completely execute. Tom will be talking about the tests more in the next chapter."

"If a business rule such as CD Rental Limit is complicated, then you would have tests that exercise just the business rule. The test scenarios for the use case would exercise two conditions – when the business rule passes and when it fails. If there was a particular risky aspect to the business rule, then you might create more test cases for the scenario."

The Procedure

"In general, use cases," Debbie states, "are more than just our joint understanding of how things should work. They also document the computer part of the workflow. If you create a user's manual for the clerks, you could just put the use case into the manual. Or you could rephrase it, so that it reads better for a non-computer savvy person. Each use case captures all of the issues for a particular operation. So it is a document that it is worth making correct."

Communication

In lean/agile development, the triad communicates more through face-to-face interactions, than through written documentation. The user stories, use cases, and acceptance tests are developed interactively. Face-to-face meetings with a whiteboard to record and display ideas are the most effective form of communication. [Cockburn02]. If they are separate, then having a video meeting with a shared desktop is an alternative.[4] Let's take a bird's eye view of how Tom, Debbie, and Cathy interact in these face-to-face meetings.

All three perform 'Active Listening' [Mindtools01]. In Active Listening, they listen to understand. If they understand, they acknowledge their understanding with a "I follow you" gesture – a nod or a verbal affirmation. They focus on what the speaker is saying, not what they are going to say next. If they need clarification, they ask for it, such as "Give me an example."

When recording ideas on the whiteboard, they practice what I term Active Writing. Recording on a whiteboard, rather than on paper provides instant feedback. When a person is recording ideas, the speaker waits till each idea is recorded before proceeding to the next. That keeps the pace reasonable. If an idea is not recorded clearly, the group can immediately suggest a correction. Ideas are clarified in person and recorded with a common understanding.

[4] You can use video conferencing sites such as oovoo.com or skype.com

When documenting ideas, they recognize that each person may have a preferred way of receiving information. Some like textual descriptions in either prose format or outline form. Others would rather view diagrams and charts than text. If necessary, information is recorded in both formats, so that both preferences can be honored.

When they are brainstorming or describing ideas, they realize that each person can have different responses. Some people get their energy from verbal discussions with other people (extroverts) [Wiki05] while others process their ideas internally (introverts). So the triad has mechanisms for allowing both to interact. They have times when people think individually and write down thoughts and when people discuss thoughts as a group.

They understand that some people like to see the big picture without getting into details (intuition) while others want to see the details (sensing). So they have both brief requirements such as user stories and detailed requirements, such as use cases. They recognize that usually progress can be made without first gathering all the details. But they acknowledge the times that work needs to stop if an important detail is unknown.

They realize that clarity is important. So they develop a common terminology. The developer and test units accept that the terms and definitions come from the business unit. The business unit understands that the ambiguous terms they use may have to be renamed in order to provide clarity.

Communication Is More Than Wordsc

Communication, even when you understand it, can be difficult. We each have our own preconceived notions as to what is clear and what is correct. I was working with a colleague on developing a PowerPoint presentation for a conference. We had gone through the slides together and had a good working understanding of what we were going to present.

A little before the presentation, I got a printed copy of the slides from him. They were printed four to a page – two rows and two columns. I looked at the printout and exclaimed that he had rearranged the slides. He looked at the printout and said that he had not.

Here's a question for you. If the first slide is in the upper left portion of the page and the fourth slide is in the bottom right portion, where should the second and third slides be located?

If you said the second should be upper right and the third should be lower left, then you would have been as surprised as I was. If you said the reverse, then you would have had no issue with the printout.

Communication is about more than just words. It's about how you organize those words.

Summary

- A use case details a user story
- A use case states the pre-conditions, post-conditions, and main course.
- A use case may have exceptions which do not allow it to successfully complete.

- A use case may have alternative ways to achieve the post-condition.
- If a use case is large, exceptions and alternatives may become user stories.
- Use cases suggest acceptance tests.
- Collaboration requires an understanding of the difference in how people create and process information.

Chapter 8

Test Anatomy

"Whoever named it necking was a poor judge of anatomy."

Groucho Marx

Tom expounds on testing (see Figure 8.1) to Cathy. He explains the basic structure of tests – given, when, then. He shows how tables can clarify the examples used for testing and gives an overview of three types of tables.

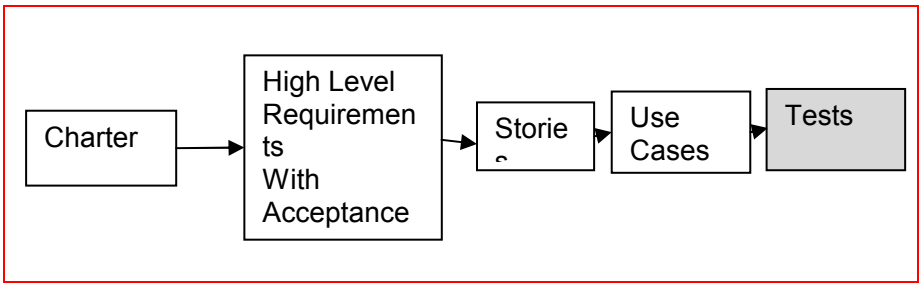


Figure 8.1 *Creating Tests*

A Few More Tests

Tom starts, "In the previous chapter, Debbie stated the most obvious tests associated with the Check-out CD use case. We've found that tests are created by all three parts of the triad. The customer unit usually comes up with the basic workflow tests and the developer and testers units come up with ones from their training and experience. As the tester, I am responsible for ensuring that there is a set of tests that is as complete as practical. From my testing experience, I can envision more tests that can be run for this use case. For example:"

Checkout Rented CD – Customer attempts to rent a CD that is already rented
CD ID Not Recognized – The CD ID is not recognized by the system

Tom continues, "Ideally, we want to come up with all of tests prior to Debbie starting to implement the story. But sometimes she discover one while coding and sometimes I realize more tests after she's completed her code."

The Text Context

Tom starts, "A system's operation is defined by its inputs and its outputs, as shown in Figure 8.2. This is a context diagram. What is external to a system is outside circle. These externalities define the context in which the system operates. An input or sequence of inputs should result in a determinable output. For example, if you input a rental for a particular customer and a particular CD, you should get a rental contract. If you input another rental for the same CD without it being checked in, you should get an error."

"The response of the system is different the second time you try to rent the CD, since the system has stored the rental information for the first check-out. Or as we like to say, the rental data has been made persistent."

"A system can store the rental data internally or external. If it stores it internally, we refer to that as changing the state of the system. If it stores it externally, it is simply another output and input to the system."

"For Sam's system, we are going to treat the data as if it were internally persistent. So we would say that the renting a CD the second time causes a different output because the state of the system is different. As you will see, part of a test involves specifying what is the current state of the system."

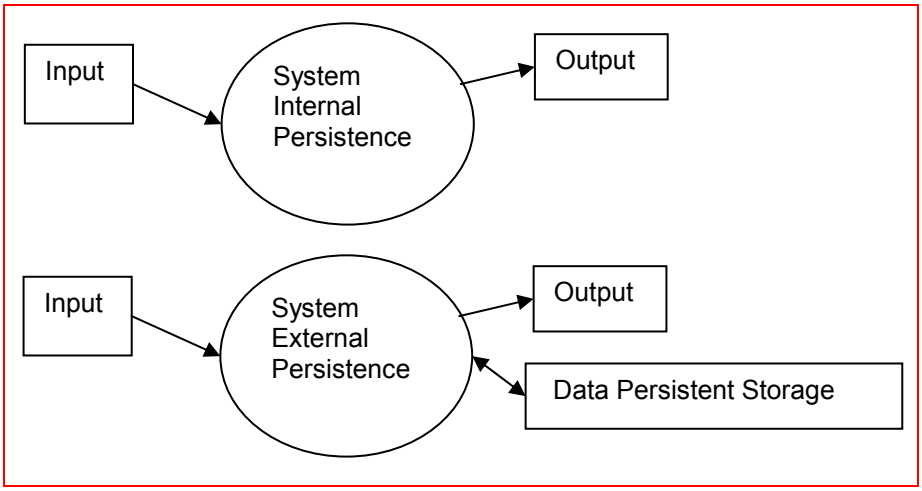


Figure 8.2 *Context Diagram*

Structure of a Test

Tom continues, "There is a basic flow to a test as shown in Figure 8.1. The test starts with setting up the state of the system. Then an event is made to happen. The test has an expected outcome of that event – a change in the state of the system or an output from the system. The test compares that expected outcome to the actual outcome of the system under test. If they are equal, the test succeeds; otherwise it fails. The flow is often shown as the following (see Figure 8.3):"

```
Given <setup>
When <event or action>
Then <expected outcome>
```

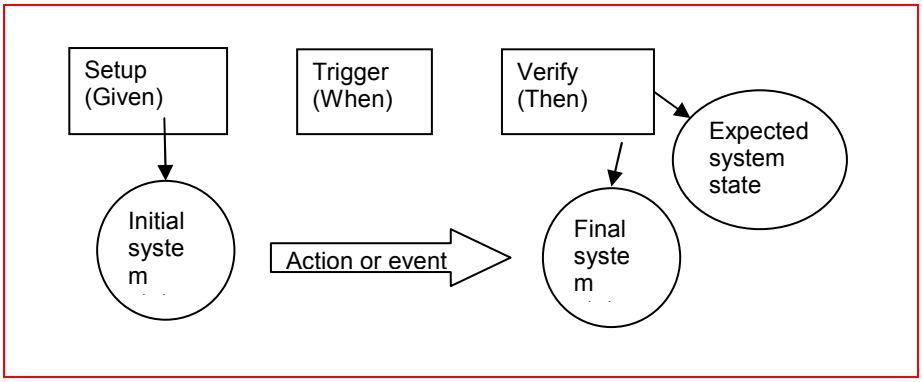


Figure 8.3 *Test Flow*

"The flow for the tests for Check-out CD would be something like:"

Given

Customer has ID

CD has ID

CD is not currently rented

When

Clerk checks out CD

Then

CD recorded as rented

Rental Contract printed

"For simple calculations, like the discount that we talked about in Chapter 4, the action is just calling some module to do a calculation. So the form could look like:"

Given <inputs>
When <computation occurs>
Then <expected outputs>

"If you recall, from that chapter, we had computations that looked like the following:"

Discount		
Item Total	**Customer Rating**	**Discount percentage?**
10.00	Good	0
10.01	Good	1

50.01	Good	1
.01	Excellent	1
50.00	Excellent	1
50.01	Excellent	5

So a single test would be:

```
Given Item Total of 10.00 and a Customer Rating of Good
When computing Discount percentage
Then output should be 0.0
```

Tom continues, "There is often an overlap between what is called a test and what we call a test scenario. In the discount example, there are six different combinations of values for the input. Each combination (or row in the table) is a test case. For something like a calculation, the group of these test cases could be referred to as a calculation test or a business rule test.

"We will see shortly a more elaborate setup. Each exception or alternative in a use case is called a scenario, because there is a different flow through the use case. Any time you have a different flow, you need a different test scenario. Each test scenario will have a different setup or a different action. A test case is a test scenario with the actual data. Because of our working background together, Debbie and I often refer to something as a test. Sometimes a test can mean plural or singular, depending on the context. We'll try to distinguish the difference when we have our discussion with you."

Tom says, "As we saw in Chapter 4, business rule tests are usually not as complicated as scenario tests. The business rules often have no initial state setup and the verification is simply comparing a single result to the expected result. The more that business rules can be separated from scenarios, the easier it is to create and run the tests."

Table Driven Tests

Tom begins, "When we write a test, we want to use the same domain language that we used in writing up the stories. The consistency keeps down misunderstandings. If we discover during the test writing that there is an ambiguity in the terms, we go back and fix up the glossary and the stories."

"Some developers and testers use what looks like regular text to specify the tests. So they might write something that looks like this:

```
Given Customer has ID and CD has ID and CD is not currently rented
When Clerk checkouts CD
Then CD recorded as rented and Rental Contract printed
```

Calculation Table

"Debbie and I have found that adding examples to tests in tabular structures gives a more consistent feel to the tests and often exposes misunderstandings between the customer and developer units. We use the text to outline what is the flow of the test. The table structure used in the discount example in Chapter 4 was:

Title			
Input Name 1	**Input Name 2**	**Result Name?**	**Notes**
value for input 1	value for input 2	expected output	anything that describes scenario
another value for input 1	another value for input 2	another expected output	

"This structure is used primarily for calculations.[5] An example of this table with only one input column might be:"

CD Rentals	
CD ID	**Rented?**
CD2	No
CD3	Yes

"The input name is CD ID and the result name is Rented. The value for the input is CD2 and the expected output is No."

"Another example of this table is the one we used for discounts. The two input names are Item Total and Customer Rating. The result is Discount percentage. The input values are 10.00 and Good and the expected output is 0."

Discount		
Item Total	**Customer Rating**	**Discount percentage?**
10.00	Good	0

Data Table

"There is a second table structure which declares that information in the system exists (or should exist). You can use some part of the name to indicate that it is a data table rather than a calculation table."

Title Data	
Value Name 1	**Value Name 2**
value for 1	value for 2
another value for 1	another value for 2

"Here's an example of customer data. This shows that there should be a customer whose name is James and whose customer ID is 007 and a customer named Maxwell whose Customer ID is 86."

Customer Data	
Name	**ID**
James	007

[5] The structure is sometimes used for actions, such as recording that a CD is rented, particular for lots of test cases for the same action. Another structure that we'll get to shortly is used specifically for actions

Maxwell	86

"The columns represent the different fields in a data record and each row represents a data record. The table does not have to correspond to any specific database table. It could represent a temporary collection of the data items. It is the user's view of how the data elements are related, regardless of how they are stored. If the table is used for setup, the data is put into the database or collection, if it does already exist. If the table shows expected values, the test fails if the data items do not exist or have different values."

"There is a variety of the data table that shows only rows which meet a certain criteria. You specify the criteria after the name. For example, if you only wanted to see customers whose Name began with 'J', you could have:

Customer Data	**Name Begins With='J'**
Name	**ID**
James	007

Action Table

"The third table structure is an action table. The easiest way to describe the table is that it works like a dialog box, although the action table can be used for many more situations. It has two different formats that are equivalent.[6] It all depends on which way you prefer to layout things."

"The first form starts with a title that represents a procedure or the name of a dialog box. The first column has one of three verbs - *enter*, *press*, and *check*. Each verb has an object that it uses. *Enter* enters data into an entry field, *Press* initiates a process, such as a Submit button on a dialog box. *Check* sees if a result is equal to an expected value.

Action Name		
Enter	**Value Name 1**	Value for 1
Enter	**Value Name 2**	Value for 2
Press	**Submit**	
Check	**Value Name 3**	Expected value for 3

"An example of the first way is:"

Check-out CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD2
Press	**Submit**	
Check	**Rented**	True

"The other layout makes the Enter and Press verbs implicit. It puts all the entry field names and the values into a single row. The Press verb is implied by the end of the row. The Check verb is used on separate rows. This layout looks like: [Martin02]

[6] Developers - this is not the exact structure for the standard Fit Action Fixture. These table examples are meant to be generic.

Action Name	**Value Name 1**	Value for 1	**Value Name 2**	Value for 2
Check	**Value Name 3**	Expected value for 3		

"An example of this layout look like:"

Checko-ut CD	**Customer ID**	007	**CD ID**	CD2
Check	**Rented**	True		

"Just like anything else in the world, there are some people who prefer the first layout and some who prefer the second. To keep things consistent, we will just use the first. "

"Some people are horizontally oriented. Others are vertically oriented. The action table can be represented horizontally, as shown in the second layout. If you are doing a repeated set of actions, using either of the layouts requires repeating the value names. So sometimes we use a table that looks like a calculation table, but it's really for actions. For example, if you checked out two or more CDs, you could have:"

Check-out CD		
Customer ID	**CD ID**	**Rented?**
007	CD2	True
86	CD1	True

"To keep things more consistent and to make a distinction between calculations and actions, we're not going to use this version for this system. However, in the real world, we often employ this type of table for repeated actions."

Test Structure With Tables

Tom starts again, "Now let's put these tables into the test structure:"

Given Customer has ID and CD has ID and CD is not currently rented

Customer Data	
Name	**ID**
James	007

CD Data		
ID	**Title**	**Rented**
CD2	Beatles Greatest Hits	No

When Clerk checks out a CD

Check-out CD		
Enter	**Customer ID**	007

Enter	CD ID	CD2
Press	Submit	

Then CD is recorded as rented and Rental Contract printed

CD Data			
ID	Title	Rented	Customer ID
CD2	Beatles Greatest Hits	Yes	007

Rental Contract			
Customer ID	Customer Name	CD ID	CD Title
007	James	CD2	Beatles Greatest Hits

"Now the Rental Contract shows the information that will be printed on the form, but not all the surrounding text. This way you can be sure that the correct information is on the contract. Later on you can decide with Sam on how the rental contract should be worded."

A Requirements Revisit

"Now that you can see how the test is structured, does it look like there is anything missing?" Tom asked.

"Yes, there is," Cathy replied. "The tables definitely make things more apparent. Every CD has a rental period. If the customer returns the CD after the end of the rental period, then we charge them a late fee. The Rental Contract should have the date of the end of the rental period. We also want the rental fee itself on this contract, but I think we cover that in another story."

"Okay," said Tom. "Let's revise the tables to include this rental period. Let me make sure of something. To get the date for the rental period end date, you add the rental period to the start date. Is that right?"

"Sure," said Cathy.

"Okay, so just a quick table to check it out both the calculation and our terminology," Tom hinted.

Calculate Rental End			
Start Date	Rental Period (Days)	Rental Due?	Notes
1/1/2010	2	1/3/2010	
2/28/2012	3	3/2/2012	Leap Year
12/31/2010	4	1/4/2011	New Year

Debbie interjects, "You can see that Tom is always thinking about the odd cases. He keeps me on my toes that way."

Tom continues, "So you want the Rental Due on the Rental Contract. I guess we should keep it with the CD so we know when it's due. I just have a feeling from the bigger picture – the Inventory Report story – that it would be a good idea. Some developer's might suggest YAGNI ("You Ain't Gonna Need It") and not do it. Since that story is coming up soon, it's okay to consider it now as part of our big picture scope."

Tom continues, “So given that we have that simple calculation correct, then what our tests needs to do is to set the current date. We do not want to have to change our test, just because the date has changed. I'll show how we set a date here and talk about it more about test doubles in Chapter 11. So the test could now read:”

Given Customer has ID and CD has ID and CD is not currently rented:

Customer Data	
Name	**ID**
James	007

CD Data			
ID	**Title**	**Rented**	**Rental Period**
CD2	Beatles Greatest Hits	No	2

Test Date
Date
1/1/2010

When Clerk checks out CD:

Check-out CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD2
Press	**Submit**	

Then CD recorded as rented and Rental Contract printed:

CD Data				
ID	**Title**	**Rented**	**Customer ID**	**Rental Due**
CD2	Beatles Greatest Hits	Yes	007	1/3/2010

Rental Contract				
Customer ID	**Customer Name**	**CD ID**	**CD Title**	**Rental Due**
007	James	CD2	Beatles Greatest Hits	1/3/2010

Tom concludes, “We’ve presented these table examples with formatting that distinguish between the column headers and the data. The formatting is not mandatory. When coming up with these on the whiteboard, you don’t bold the headers. We often take pictures of the whiteboard once we're done gathering these, transcribe them into tables, and have the customer review them to make sure no errors crept in."

Summary

- The structure of a test is
  - Given <setup>
  - When <action or event>
  - Then <expected results>”
- For calculation tests, the structure is
  - Given <input>
  - When <calculation occurs>
  - Then <expected results>
- Three types of table are:
  - Calculation – gives result for particular input
  - Data – gives data that should exist (or be created if necessary)
  - Action – perform some action

Chapter 9

Tests in Action

"I did then what I knew how to do. Now that I know better, I do better."

Maya Angelou

The triad creates tests for the exceptions in a use case. Tom explains the levels at which the tests are run. Tom discusses that not every condition should be automated. Debbie shows how early implementation can give quick feedback on meeting the charter's objectives.

Implementing More Tests

Tom continues, "We've finished off the test for the main course of the 'Check Out CD' use case in the previous chapter. So we can start on the details of the other tests. First let's create a test for the scenario 'Checkout Rented CD' – Customer attempts to rent a CD that is already rented. The change is in the setup. And the expected result is an error message when the rental is attempted. So the test looks like the following:"

Given a CD that has already been rented

CD Data			
ID	**Title**	**Rented**	**Rental Period**
CD2	Beatles Greatest Hits	Yes	2

Customer Data	
Name	**ID**
James	007

When a Customer attempts to rent the CD, then an error message is displayed

Checkout CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD2
Press	**Submit**	
Check	**Error Message**	CD Already Rented

"You can see that the Customer Data is the same. We could put the Customer Data table into a common location and reference it from the other tests. There is a tradeoff between using a common setup and redoing the setup for every test. We'll talk about that later in Chapter 20. The difference in the setup in this case is that the CD is already rented. The difference in the result, which is put as the final step in the action is that an error is produced. The 'Check' word verifies that the 'Error Message' is reported as 'CD Already Rented'. We don't have a contract printed out since the error occurred."

"Another test scenario is 'CD Rental Limit – customer who has three CDs and rents another one'. We need to setup the situation where a customer has rented three CDs. Without repeating the Customer Data, the setup looks like: "

Given a customer who has rented the CD limit of three

CD Data	**Customer ID = 007**		
ID	**Title**	**Rented**	**Rental Due**
CD2	Beatles Greatest Hits	Yes	1/3/2010
CD3	Lucy Michelle Hits	Yes	1/4/2010
CD4	Janet Jackson Hits	Yes	1/5/2010

"We included one additional aspect in this setup. It's 'Customer ID = 007'. What this means is that this table reflects the rentals for which the Customer ID is 007 or James. That way we do not have to duplicate the '007' in every row for the Customer ID column. Also, if the CD data in the system had rentals for customers other than '007', this table only reflects those for '007'. Now you might notice that Rented is also 'Yes' for every row. So we could move that up to the first line as well.

*** The tables are a little wide here. I'm not sure how to reduce them or whether it's just a display artifact.

Given a customer who has rented the CD limit of three

CD Data	**Rented = Yes**	**Customer ID = 007**
ID	**Title**	**Rental Due**
CD2	Beatles Greatest Hits	1/3/2010
CD3	Lucy Michelle Hits	1/4/2010
CD4	Janet Jackson Hits	1/5/2010

When he attempts to rent another CD, then an error message is displayed.

Checkout CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD5
Press	**Submit**	
Check	**Error Message**	CD Rental Limit Exceeded

"In this case, the Error Message is different from the previous test. If I hadn't work with Debbie as much as I have or if this story had risk associated with it, I might also try out a condition where two CDs are rented to see if the customer is correctly allowed to rent a third. Or I might rent four CDs right in a row and make sure that the error occurs on the fourth CD. Or more. I think you get the picture."

In a financial application, one customer wanted the tester to add one more test scenario. In the application, the net worth of the corporation was computed every day. The net worth calculation depended on the current Federal Reserve inter-bank rate. The customer said, 'I want you to see how the application reacts if the inter-bank rate is not available. I can think of a instance where Federal Reserve goes broke and so we can't get the rate."

The tester replied, "Then I think we'll have a few other things to worry about in that case. Without the Federal Reserve, our net worth would be zero."

Cathy's Test

Cathy exclaimed, "I think I've gotten the idea now. Sam and I were thinking of another business rule. We won't let a customer rent another CD if they have one that is late. So based on your examples, here's what I think the test should look like this:"

Given the Customer has a rental that has not been returned by the due date

Test Date
Date
1/4/2010

CD Data				
ID	**Title**	**Rented**	**Customer ID**	**Rental Due**
CD2	Beatles Greatest Hits	Yes	007	1/3/2010
CD3	Lucy Michelle Hits	No		

When they attempt to rent another CD
Then notify them they have a late rental and they cannot rent the CD

Checkout CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD3
Press	**Submit**	
Check	**Error Message**	Customer Has Late

	Rental

Don't Automate Everything

Tom starts off, "Cathy, we've created a couple of exceptions for entering a bad customer id." Debbie could program this into the system. She could keep track of the number of times a bad customer id was entered and put up an appropriate error message. I'd have to write some tests to ensure that was coded correctly. Instead, initially these exceptions could just be handled by manual instructions to the Clerk. They could be something like:"

> If the system responds with a bad customer id, try re-entering the id. If you try it a second time and it does not work, then make a copy of the customer's driver's license and fill out a rental contract manually for the customer to sign.

"Now we just have to write a test for one bad customer id."

> Given that we have all valid customers in our Customer Data

Customer Data	
Name	**ID**
James	007
Maxwell	86

> When a Customer ID is entered that is not valid
> Then inform the clerk that the ID is not valid

Checkout CD		
Enter	**Customer ID**	99
Enter	**CD ID**	CD3
Press	**Submit**	
Check	**Error Message**	Customer ID Invalid

"The cost of implementing and testing for the number of bad entries is probably not justified by any business value. Part of our job is not just to deliver software to you, but to deliver software that delivers business value."

Multi-level Tests

Tom starts off, "The tests we created can be used on multiple levels within the system. For example, the CD Checkout Test can be applied at the user interface level or the mid-tier (see Figure 9.1). If we apply the test at the mid-tier, we check that the functionality works in Debbie's code. Once we design the user interface, we test that the user interface is coupled properly to the correct functionality in the mid-tier."

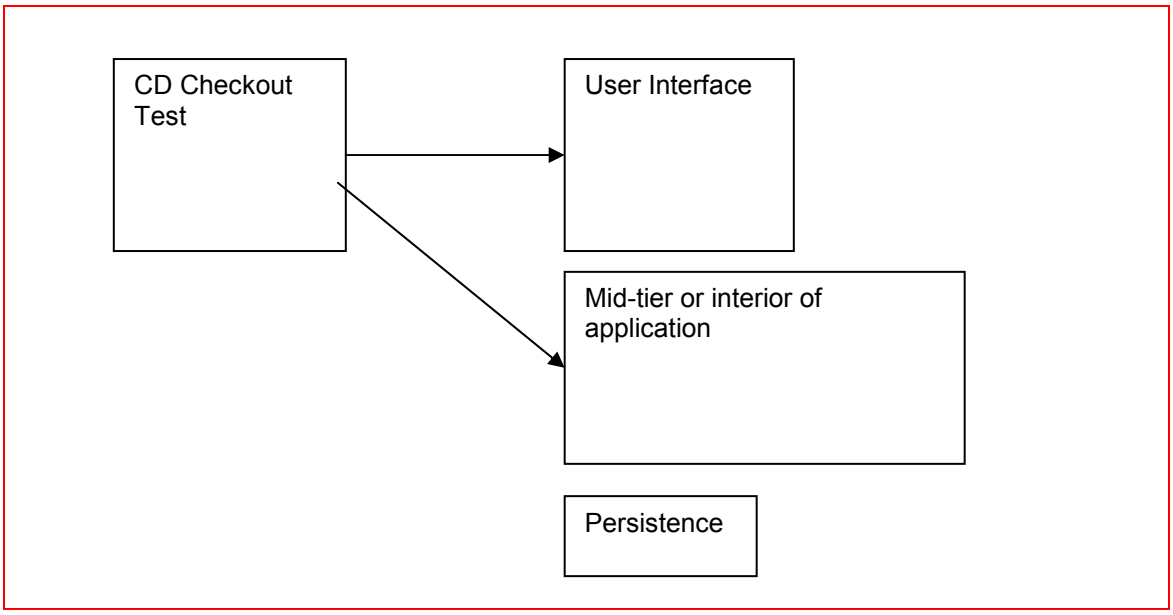


Figure 9.1 *Multi-layer Tests*

"There are some tests that we run just against the mid-tier, such as the Calculate Rental End (see Chapter 8). The results of that calculation show up in an output in the Rental Contract (see Figure 9.2). I've added a screen (CD Screen) based on the setup and expected outcomes. We could use this additional screen just during testing, but it seems that it also might come in handy when the application is in production.

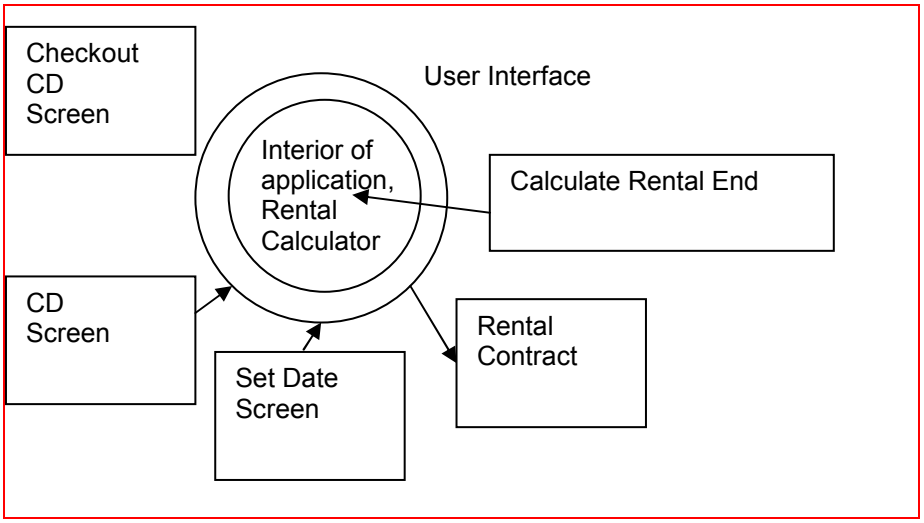


Figure 9.2 *Tests for different layers*

The Check-out CD Screen might look like what is shown in Figure 9.3.

***Do as line art

Check-out CD		
	Customer ID	007
	CD ID	CD2
	Submit	

Figure 9.3 *Tests for different layers*

The CD Screen could look like what is shown in Figure 9.4.

***Do as line art

CD Data		
	ID	CD2
	Title	Beatles Greatest Hits
	Rented	Yes
	Customer ID	007
	Rental Due	1/3/2010
	Rental Period	2

Figure 9.4 *The CD Screen*

"The Calculate Rental End test goes into the heart of the application. You probably would not use it in regular operations. So we are not going to create a user interface for it, just use a test that goes to the mid-tier. However, we do need the ability to set the date for the application, not for the entire computer. Otherwise other programs may be affected. So we could either have a Set Date screen that allows the tester to manually set the date or we could have an input at the start of the program (often called a command line parameter) that sets the date."

"The reason why we run Calculate Rental End to the mid-tier is that we can run many test cases on this business rule without having to invoke the user interface. As we talked about earlier (Chapter 4) tests run directly to the mid-tier allow execution of lots of test cases without getting carpal tunnel syndrome."

"We do not need to run all the test cases through the user interface. If Debbie, you, and I agree that there is little risk for a particular aspect of a story, then it makes more sense for us to concentrate our time elsewhere. We will run at least one case for each scenario which causes the user interface to generate a different display, such as an error message."

The User Interface Tests

Tom starts off, "After Debbie finishes the first version of the user interface and tests it against the acceptance tests, we can start to get feedback from you and the Clerks. The user interface may change dramatically based on user comments. The order and position of the two input fields might change. Or we might not have a submit button on the check-out screen. When both fields are filled in, the rest of the rental process would commence."

"For each of the error messages that appeared during the check-out tests (such as 'Customer has a late rental'), Debbie will create an indication on the display. The message could appear on the entry screen or the message could appear in a separate dialog box. The error could create a loud beep or just a quiet ding." She'll check with Cary, Mary, and Harry to see how they would like it to show up. We will discuss later (Chapter 14) ways to capture tests for displays."

Have We Met Objectives?

Debbie starts off, "Once we have a user interface for Check-out CD, we can see how the check-out time compares to the objective stated in the project charter. Remember that measure is 50% less time. (see Chapter 5. We are going to start with the easiest way to implement the check-out screen. The benefit of this approach is that it requires no hardware. However, it does take more time and introduces the possibility of errors, even if appropriate check digits are incorporated in the IDs. If we meet the objective, then we are done."

"If we come close to the 50%, we have a system that is faster than before. You and Sam can decide whether there is a business reason for spending more money to reach the 50% faster measure. If there is little financial justification, you may want to revise the objective."

"If the time is far off, then we can investigate ways to cut it down. We could add a handheld bar code scanner for either or both ids. Chances are pretty good that customer might forget to bring their customer card with a bar code. So we might have to figure the potential time savings for just scanning the CD. And we need to take into account unreadable bar codes. If the handheld scanner isn't fast enough, we could look at an in-counter scanner."

"If the bar code scanner doesn't look like it will be fast enough, well, Debbie has been dying for an opportunity to try out those new RFID (Radio Frequency ID) microchips. With an RFID embedded in the CD case and one in the customer identification card, you could check out the customer as he walked past the clerk's desk." Debbie's eyes light up at the mention of RFID and says, "I'm pretty sure that's overkill for this size operation. But when Sam ramps up the marketing for this place after the software is developed, it might be a thing to try."

"If we had a larger system to measure, we might record a log file that monitors the speed and correctness of entries. The time delay due to errors or slowness could be converted to dollars based on some conversion ratio. Unfortunately, the negatives caused by the delay or customer impatience are harder to measure. When the dollars lost due to delay justifies the cost of a scanner, we can upgrade the system."

Summary

- Create a test for each exception and alternative in a use case
- Do not automate everything
- Run tests at multiple levels
- Create a working system early to check against objectives

Chapter 10

Another Story

""Life affords no higher pleasure than that of surmounting difficulties, passing from one step of success to another, forming new wishes and seeing them gratified."

Samuel Johnson

The triad meets again to discuss another story. Cathy discovers how stories can emerge from the details. Tom explains how tests should not be redundant. Cathy is amazed at the boundary tests that Tom creates.

Another Story

Debbie began, "It looks like the first story is done, at least according to functional acceptance tests. I will work with Mary, Larry, and Cary as they are available on how the user interface screen should appear. So it's time to start on another story. Normally I'd do the Check-in story next, since that's related to Check-out. But since the acceptance tests for that story follow along the same lines as the tests for Check-out, I'll start on a different story that can demonstrate other issues. So we'll discuss Charge Rentals. As you may recall, the story was:"

> Charge Rentals - As the Finance Manager, I want to submit a credit card charge every time a CD is rented so that the store does not have to handle cash.

Debbie states, "When we initially discussed the story, we thought about breaking it into two stories:"

> Compute Rental Fee.
> Check out CD. See if rental charge is correct. See if credit charge matches rental charge.
> Process Charge
> See if charge made to the credit card company. Check that bank account receives money from the charge.

Debbie continues, "I'd keep these as two separate stories, based on the tests. The tests for the first one relate to computing the correct charge. The tests for the second one revolve around transactions and interfaces with third parties. So Cathy, could you explain the details for the first story?".

Cathy answers, "Sure. Sam and I created three categories of CDs – Regular, HotStuff and GoldenOldie. We have different rental rates for each category. They are:"

CD Rental Rates
Regular $2 / 2 days plus $1 / day
GoldenOldie $1 / 3 days plus $ .50 / day
HotStuff $4 / 1 days plus $6 / day

Debbie asks, "To make it clearer, I'd like to put the values for these rates in a table." Cathy replies, "Sounds fine to me." Debbie draws the following:

Rental Rates			
CD Category	**Rental Period Days**	**Rental Rate**	**Extra Day Rate**
Regular	2	$2	$1
GoldenOldie	3	$1	$.50
HotStuff	1	$4	$6

"Did I get that right?", Debbie queries. "In particular, did I name the columns what you call the concepts? The Rental Rate is a base rate and the Extra Day Rate is for days over the Rental Period Days?"

Cathy replies, "That's the way it works now. I know we discussed a late fee in our original talks. Sam and I agreed we should just make a single charge when the CD is returned, rather than two separate charges."

Debbie asks, "Do these rates ever change?"

Cathy answers, "Not too often. But I obviously would like the ability to change these rates."

Debbie comments, "Let's make up another story for that. As you can see, there are often new stories that emerge when we get to the details. The story could be:"

As the finance manager, I need to modify the rental rates.

She continues, "We'll get everything done for this set of rates. I have the big picture that the rates will change. When I program this story, I'll code it so that the effort to add modifiability won't be a big deal. If it would take me a lot of work to make the code easy to change, then I'd code what I needed now and make the alterations later. When I eventually add modifiability, I'll know that my alterations did not affect the original story, since there are all the acceptance tests we are going to develop around that story."

Tom interjects, "Now that you two have got the future solved, I think we have enough information to create a set of tests for the rental fee calculation. Here's my initial thought:"

Rental Fees			
CD Category	**Rental Days**	**Rental Fee?**	**Notes**
Regular	2	$2	
Regular	3	$3	Extra day
Hot Stuff	2	$10	Extra day

Cathy replies, "That doesn't seem like enough. You don't have an example for Golden Oldie."

"We can add one," Tom responds. "We're going to have a test for the rate table you proposed. So we'll have already made sure that the Rental Period Days and so forth are already correct for Hot Stuff.

The test cases in the table show that we can compute the Rental Fee correctly for a normal and an extra day rental. If we had fifty different CD Categories, then repeating the same calculation for each one would be redundant. We don't want to over test a low risk situation, such as a simple calculation. Besides, if we had all fifty in this table and you changed the formula, we'd have to change all fifty results. But since there is only one more category, we'll add it to the table. We can even suggest a reason for adding it. So the additional row looks like:"

Rental Fees			
CD Category	**Rental Days**	**Rental Fee?**	**Notes**
Golden Oldie	4	$1.50	Extra day shows cents in the rental fee

Tom continues, "I'd like to ask about some more test cases. Since I always think about the boundary conditions, let me see if I've interpreted your rule correctly:"

Rental Fees			
CD Category	**Rental Days**	**Rental Fee?**	**Notes**
Regular	1	$2	
Regular	100	$100	
Regular	0	$2	

"Wow!" exclaims Cathy. "You really do have an active mind. I never even thought about those last two test cases. That one charging someone $100 for a rental seems right according to your calculations. But that doesn't seem right for the business. I think we need to cap the amount the rental fee to the price of the CD. It will take a little bit of time for Sam and I to get together to determine how that should work – whether a rental that goes on for a number of days is automatically terminated and the CD is sold to the customer or whether we give the renter a call or so forth. I've got the story idea now. I think there may be a couple of stories:"

As the Finance Manager, I want to limit the rental fee for a rental.
As the Inventory Maintainer, I want to be able to handle a rental that goes on for a long time.

Tom chirps in, "By George, she's got it! By George, she's got it!" Cathy grinned back.

Cathy continues, "So what do you mean by that test for 0 Rental Days? We never rent anything for 0 days. We wouldn't make any money doing it."

Tom replies, "It's unclear to me how you determine Rental Days. Is it a twenty-four hour period? What if someone checks out the CD and then immediately returns it? How long is that?"

Cathy smiles, "You are really being picky. I guess I need to be more precise so that you can give me exactly what I want. If a rental is returned by 11:59:59 p.m. on a particular day, we count that as being returned on that day. We charge the Rental Rate for anything that is returned on or before the Rental Due Date. So it doesn't matter if they return it on the same day. It's still charged the full Rental Rate."

Tom answers, "Great. I might make up a table that gives examples of what you just said. But I can't see any ambiguity. I had that one place where they were using time periods based on minutes."

Cathy says, "I'm afraid to ask, but can you show me how that created a problem?"

Debbie interjects, "Tom loves to tell this one. Since it's an instance of how to make sure the team is dealing with all the exceptions, I'll let him go on."

"Thanks," says Tom. "So this was a case of where they needed to calculate days, hour, and minutes. I came up with the following table:"

Calculate Time Period							
Start Date	**Start Time**	**End Date**	**End Time**	**Days?**	**Hours?**	**Minutes?**	**Notes**
1/1/2008	12:01 AM	1/2/2008	2:04 AM	1	2	3	
2/28/2008	12:01 AM	3/1/2008	3:05 AM	2	3	4	Leap Year
11/2/2008	1:59 AM	11/2/2008	1:01 AM	0	0	-58	Do you know why ??

"What's that last one?" Cathy inquires. "You can't have an end time that is before the start time."

Tom answers, "It's when daylight savings time ends. There is a small window between 1 am and 2 am. If the start falls within this window and the stop occurs within one hour of setting the clock back, it's possible to get negative time. Even if you have a start before this window and a stop after this window, you get one less hour than exists in reality."

"Okay, so what did you do?" Cathy asks.

Tom responds, "The customer said that they didn't want us to spend the time figuring out how to handle it. The situation would occur so rarely it wasn't worth trying to solve it. So we just limited the number of minutes not to be less than zero.

The Check-In Story

Debbie starts, "Let's see how all the business rule calculations fit into a check-in scenario. Normally we'd start with a use case that exceptions, but I'd like to get directly to the point. Here's an acceptance test to show how the business rule result is incorporated in it."

Given a CD is rented to a customer

Name	**ID**
James	007

CD Data					
ID	**Title**	**Rented**	**CD Category**	**Customer ID**	**Rental Due**
CD3	Janet Jackson Hits	Yes	Regular	007	1/3/2010

When the clerk checks in the CD

Test Date
Date
1/4/2010

Check-in CD		
Enter	**CD ID**	CD3
Press	**Submit**	

Then CD recorded as not rented and the correct rental fee is computed

CD Data					
ID	**Title**	**Rented**	**CD Category**	**Customer ID**	**Rental Due**
CD3	Janet Jackson Hits	No	Regular		

Rental Fee				
Customer ID	**Name**	**Title**	**Returned**	**Rental Fee?**
007	James	Janet Jackson Hits	1/4/2010	$3

Debbie continues, "We don't have to run through all the combinations that we tested in the rental fee computation. One will do. We could do a second one just to ensure that the check-in flow is correctly tied to the rental fee computation and it wasn't just a luck that we happened to get the right rental free. We should run some negative test cases, such as trying to check-in a CD that hasn't been rented or one that does not exist in the CD data. The cases essentially follow the same form as the Check-out test cases. And we don't want to be redundant."

Summary

- Creating acceptance tests can yield additional ideas
- Break acceptance tests into ones for business rules and ones for flow
  - Business rule tests verify all combinations
  - Flow tests confirm where business rules alter flow
- Determine the edge cases and how the system should respond

Chapter 11

The World Outside

"You cannot always control what goes on outside. But you can always control what goes on inside."

Wayne Dyer

The triad works on stories that involve interfaces to external systems. Debbie explains why developer stories are created. Tom explicates on test doubles and mocks. Cathy learns how story maps organize stories into workflows.

External Interfaces

Debbie starts the meeting, "Now that we've determined correctly how much to charge the customer, let's move onto the Process Charge story. Our high-level tests for this were:"

Process Charge

- See if charge made to the credit card company. Check that bank account receives money from the charge.

"Let's start by drawing a context diagram for this process. Cathy, can you explain how the charging works?"

"Sure," Cathy answers. "I've talked with my bank and the credit card processor. The rental system needs to send a charge to the credit card processor, like our current charge card reader does. The processor sends back a confirmation that the charge is proper or a denial. At the end of the day, the processor makes a bank transfer for all the charges during a day less any charge backs and the processing fee. Don't get me started on the size of that processing fee. Anyway, I can go on-line and verify the transfers that were made during the previous days. If I need to, I can get a listing of all the charges that were made from the credit card processor. I can also confirm with my bank to see that the transfer was received."

Debbie said, "Here's a diagram in Figure 11.1 of my understanding of what you said. Am I right? "

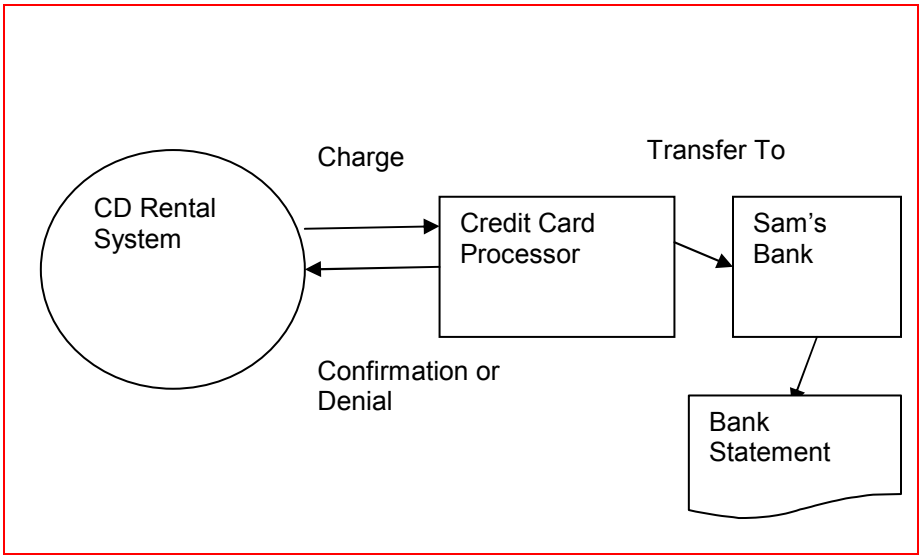


Figure 11.1 *Credit card charge processing*

"That looks good to me," Cathy replies.

Tom pops in, "So to make sure that we have correctly communicated with the processor, Cathy, you need to get a list of charges that we sent to the processor. Is that right?"

"Right on," Cathy says.

"And since you're on a roll, why don't you come up with an acceptance test for a part of this process?" Tom continues.

"Sure," Cathy replies. "It seems like there are two acceptance tests. The first is to verify that the credit card processor got all the transactions that the rental system said they sent. And the second makes sure the bank got all the transfers that the processor said they sent. So the first one would be:"

Given a list of credit card charges that the rental system sent for a day:

Credit Card Charges From Rental System	Day = 1/1/2010		
Card number	Customer Name	Amount	Time
4005550000000019	James	$2	10:53 AM
4111111111111111	Maxwell	$10	10:59 AM

When I request a list of charges for the day from the credit card processor

Request Rental Charges From Processor		
Enter	Date	1/1/2010
Press	Submit	

Then the charges should match the list from the rental system.

Credit Card Charges From Processor	Day = 1/1/2010		
Card number	Customer Name	Amount	Time
4005550000000019	James	$2	10:53 AM
4111111111111111	Maxwell	$10	10:59 AM

Cathy continues, "As finance manager, I've gotten this report many times. I've only been concerned with the dollar numbers. The three of us should take a look and see if there is any other thing on the report that you might need to know about. There is a column for transaction id."

Tom replies, "That would be great. We could run this test just as an acceptance test for the system. In that case, we would do a manual comparison of the two lists. We could incorporate the comparison into part of daily process. If we did that, Debbie could create a way to compare the information you downloaded to the list that the rental system generates."

Debbie interjects, "I don't think it should take long to program. You'll have to decide whether it's worth the money or not. Have you ever had a problem with a charge not showing up?"

Cathy answers, "There may have been one or two, but it's never been an issue that crossed my mind. The effort of calling up the credit card processor to check on a $2 charge isn't really worth it. So I think we can put the Automatic Comparison of Submitted Charges to Processed Charges story on hold. That sounds like such a great acronym though - ACSCPC. I could tell Sam that we ACSCPC every day."

"Oh, I forgot," Cathy exclaimed. "What about charge backs? How should we handle those?"

Debbie replied, "We can give it to you in whatever form is easiest for you to match up. If the transactions from the credit card processor are listed separately as charges and charge backs, we'll give you two lists. If they are listed on one list, say sorted by the time, then we'll give you one list. Our job is to create a system that makes it easy for you to do your job."

Tom queries, "Cathy is there another test you need to apply to the flow?"

Cathy replies, "There should be one to verify that a transfer was made each day and that the bank received it. This step follows after the first one. So the test would be:"

Given the charges processed by the credit card processor

Credit Card Charges From Processor	Day = 1/1/2010		
Card number	**Customer Name**	**Amount**	**Time**
4005550000000019	007	$2	10:53 AM
4111111111111111	86	$10	10:59 AM

When the bank statement is checked the next day for transfers made

Request Transfers From Bank		
Enter	**Day**	**1/1/2010**
Press	Submit	

Then there should be a transfer for the total of the charges

Transfers Received By Bank	Day = 1/1/2010
From	**Amount**
Credit Card Processor	$12

Cathy continues, "I do this every now and then for a whole set of days. In the past few years, I've never seen a transfer for the amount absent from the bank statement. There was one time it was off by a day, so there were two transfers on the same day. So there is definitely no need to automate this."

Cathy concludes, "This whole idea of charging our customers automatically instead of handling cash is really appealing. We may lose a few customers who don't want to have their rentals appear on their credit card statements. I know of one whose spouse would become really angry, to put it mildly, if the amount of money spent on CD rentals became known. But the savings in the Clerk's time in collecting money and balancing a cash drawer, my time in taking deposits to the bank, and the insurance costs of not having money on the premises will more than make up for any lost rentals. Please get going on this story pronto."

The Internals

Debbie says, "I think we've got the external processing down for this story. Let's take a look at the internal processing. From the results of the Check-in story (Chapter 10) we would have a list of rentals fees that are charged for a particular day. This list becomes the input for the next step -- Submit a charge to the credit card processor. So one test for this step is:"

Given the customer has a credit card number and has a CD rented

Customer Data		
Name	**ID**	**Credit Card Number**
James	007	4005550000000019

CD Data					
ID	**Title**	**Rented**	**CD Category**	**Customer ID**	**Rental Due**
CD3	Janet Jackson Hits	Yes	Regular	007	1/3/2010

When the CD is returned

Test Date
Date
1/3/2010

Check-in CD		
Enter	**CD ID**	**CD3**
Press	Submit	

Rental Fee					
Customer ID	**Name**	**Title**	**Rental Fee?**	**Return Day**	**Return Time**
007	James	Janet Jackson Hits	$2	1/3/2010	10:53 AM

Then submit a charge to the credit card processor at that time.

Credit Card Charges				
Card number	**Customer Name**	**Amount**	**Date**	**Time**
4005550000000019	James	$2	1/3/2010	10:53 AM

A Developer Story

Debbie starts off, "So far we can create a charge and we can confrim that the charge is received by the credit card processor. There is still one missing piece – having the rental system actually submit the charge. This is a technical issue. The credit card processor has coding standards and protocols on how to submit a charge and what are the messages transmitted between a merchant's system like yours and their system."

"If developing the message flow between the two systems was a really quick and easy thing to do, then that would just be a task that I would do as part of implementing the associated story. But I think it will take a bit more effort. So I'm going to create a developer story. As we discussed before, the main reasons for creating a developer story is that the story represents a significant effort or that it may be implemented by another team."

"Since it's a developer story, it's incumbent on the developer to supply acceptance tests. There are at least two acceptance tests that I can imagine. I'm sure Tom will come up with many more. They are:"

Given a valid credit card charge
When the charge is submitted to the credit card processor
Then a charge confirmation is received

Given an invalid credit card charge
When the charge is submitted to the credit card processor
Then a charge denial is received.

"Of course there are corresponding stories for charge backs or charge reversals. I don't want to be redundant. I think one example of a developer story gives you the idea.."

"Before I write up some detailed test cases, I need to obtain some detailed information from your credit card provider," Debbie continues. "While you guys go to lunch, I'll look at their website."

When Tom and Cathy return, Debbie has the following up on the board.

Given a valid credit card charge

Card Charge									
Customer Name	**Street Address**	**City**	**State**	**ZIP**	**Charge Identifier**	**CC Issuer**	**CC Number**	**Expires**	**Amount**
James	36500 Somewhere Street	Anchor Point	AK	99556	Sam CD Rental Return 1-3-2010	Visa	4005550000000019	01/2020	$1.00

When the charge is submitted to the credit card processor

- Contact website, submit properly formatted charge

Then a charge confirmation is received

- Receive confirmation with this data

Transaction Receipt			
Transaction ID	**Amount**	**Charge Identification**	**Result**
123456789012345	$1	Sam CD Rental Return 1-3-2010	Accepted

"And here's one for a rejection"

Given a invalid credit card charge

Card Charge								
Customer Name	**Street Address**	**City**	**State**	**ZIP**	**Charge Identifier**	**CC Issuer**	**CC Number**	**Expires**
James	36500 Somewhere Street	Anchor Point	AK	99556	Sam CD Rental Return 1-3-2010	Visa	4111111111111111	1/2020

When the charge is submitted to the credit card processor

- Contact website, submit properly formatted charge

Then a charge denial is received

- Receive denial with this data

Transaction Receipt				
Transaction ID	**Amount**	**Charge Identification**	**Result**	**Reason**
123456789012346	$1	Sam CD Rental Return 1-3-2010	Rejected	Card not on file

Debbie continues, "I found there a lot of reasons why a card can be rejected and a lot of other results. Some of them are for reasons such as the expiration date being in a bad format. Those types of issues Tom and I will deal with. They are standard programming concerns. I've come up with some results that call for a business decision. I may come across a few more when I get into the details. "

Rejection Reasons
Card number not on file
Contact the financial institution
Expired credit card

"You need to decide on what to do in each of these cases."
The triad works through the options and comes up with the following actions:"

Credit Card Charge Rejection Reasons		
Result	**Action**	**Notes**

Card number not on file	Inform customer Get another card info	Generate a dialog box on check-in Generate an email to Cathy
Contact the financial institution	Do not inform customer Make person look up at security camera Put up message to call police	Card may be stolen
Expired credit card	Inform customer Get another card info	Generate a dialog box Generate an email to Cathy

Debbie continues, "Tom and I will come up with tests that generate all these results. Details will need to be gathered on the wording of the dialog boxes and email messages. But those are just display concerns."

Tom starts off, "Debbie, what happens if the network goes down in the middle of processing a credit card transaction. You and I both know that periodically the Internet seems to come to a grinding halt, which is the equivalent of going down."

Debbie replies, "I'll just queue up the charges and submit them when it does come back up."

Tom answers, "What if it doesn't come up for a full day?"

Debbie counters, "We'll just send them when it does. I don't think we'll have to make many changes in the tests we've come up with. We may have to change our assumption that the date of the charge is not the same as the date the CD was returned. [7]

Test Doubles and Mocks

Cathy has a burning question. "How are you going to run all these tests? Are you going to use your credit card number? How can you make sure that a credit card is rejected for a particular reason?"

Debbie answers, "One way to do it is to use Tom's cards. He's maxed out on some of them. But the banks might get after Tom for trying to use those credit cards. So we will use what many developers call a test double [Meszaros01]. A test double is something that stands in for a real system when tests are being run. It comes from the idea of a double that stands in for the real actor when shooting a movie."

"A test double encompasses a couple of other concepts that you might hear Debbie and I or other developers throw around. They are mock, stub, and fake. [Craig01]. The mock term comes from Alice in Wonderland by Lewis Carroll. You may remember the line:"

"Once," said the Mock Turtle at last, with a deep sigh, "I was a real Turtle."

Debbie continues, "I'm not going to get into the differences and details between these three terms. That's something that developers love to discuss on blogs. The key is that using test doubles makes a system easier to test. The credit card processor provides a test double. Instead of connecting to the real credit card system, you connect to the test double they provide. The test double accepts credit card charges and returns confirmations just like the real system."

"To get the test double to return different results, you send it different combinations of values. For example, with your processor, you send a charge for the credit card number 4111111111111111. This causes a charge to be rejected."

[7] That is left as an exercise for the reader.

"If there wasn't already a test double, I would write one myself. In fact whenever there is some external interface to a system, I usually create a test double. In this case, as long as I can have all the different results sent back to me, I don't need my own test double. I haven't checked, but I'm sure there is something I can send that would create a result that puts up a dialog to call the police. If not, I bet one of Tom's cards would do that."

"To really test the complete system, we do need to make a few credit card charges all the way from the return of a CD through seeing it on the credit charges processed list. We'll use your card for a good one and use an invalid number to see if things are rejected."

What Is Real?

Tom is explaining to Cathy about real worlds and test worlds. "A system in production – 'the real world' – interacts with many things outside itself. It may ask an external service for information or to do a calculation or to perform an action. Events may occur at random times and in random sequences that require a response from the system. In production, we do not have control over these external interactions. But in test, we may need control so that the same test case can be performed over and over again and still get the same expected result. "

"An external service may always provide the same information every time we request it (in programmer's term, we call that idempotent). Even so, we may want to create a test double for it so the tests run faster. Let's look at the context of our system in Figure 11.2."

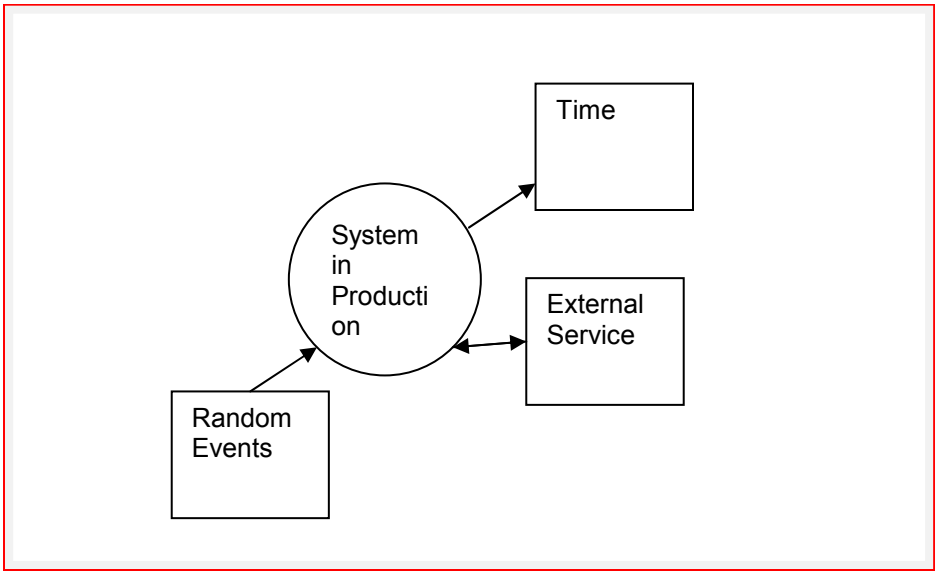


Figure 11.2 *System Context*

"We've already seen how we need to control time in order for the tests to run the same. The test double that the credit card processor provides allows for repeatable tests. If there were random events that the system had to respond to, we would create a test double for them as well."

"Just to give an example, suppose we were doing check-outs and check-ins via some kiosks. There could be several clerks all accessing the system at about the same time. We want to simulate a sequence of check-out and check-ins from these kiosks. We could create a test double that generated something like the following sequence. This is a series that occurs in quick succession. This would see if the code can handle it."

Rental Sequence				
Operation	**CD ID**	**Customer ID**	**Date**	**Time**
Check-out	CD7	99	1/1/2010	9:01:01 PM
Check-out	CD4	99	1/1/2010	9:01:02 PM
Check-out	CD5	007	1/1/2010	11:01:01 PM
Check-out	CD2	86	1/1/2010	11:01:02 PM
Check-in	CD7		1/1/2010	11:01:03 PM
Check-out	CD3	007	1/1/2010	11:01:04 PM
Check-in	CD4		1/1/2010	11:01:05 PM

Story Map

Debbie chirps up, "For the entire check-in workflow, there are several stories. Each story is related to the others by being a step in the workflow. Some stories are more detailed requirements within a particular step. We use a story map [Hussman01], [Patton01] to keep track of all these stories.

Debbie continues, "A story map organizes stories into a flow. For an activity, you usually have multiple steps as shown in Figure 11.3. The steps are usually performed sequentially, from left to right. Underneath each step are the stories associated with that step. To get through the flow, you need to have an implementation of at least one story associated with each step. The other stories provide handle exceptions or provide alternatives, as we talked about in our discussion on use cases in Chapter 6 You can place the highest priority story at the top in each column. When you have a tested implementation for each of these, you then have an implementation for the entire activity you can test. "

Activity Name		
One Step	**Another Step**	**Still Another Step**
Story For One	Story For Another	Story for Still Another
Other Stories For One	Other Stories For Another	Other Stories For Still Another

Figure 11.3 *Story Map Template*

"Since we have a lot of stories for check-in, let's put them onto a story map. This gives us a better way to relate the stories together and to make sure that we can span an entire workflow with at least the main course of each story."

Check -in Workflow			
Check-in CD	**Compute Rental Charge**	**Process Charge**	**View Report**
Check-in	Compute rental charge for single rate	Process normal charge	List credit card charges
	Compute rental charge for multiple rates	Handle declined credit cards	

Figure 11.4 *Check-in Story Map*

Summary

- Create acceptance tests for external interfaces
- Developers can create acceptance tests for internal processing functionality
- Use test doubles or mocks for external interfaces to simplify testing
- Create story maps to organize stories for workflows

Chapter 12

What Is Done and To Do

"May you have the hindsight to know where you have been, the foresight to know where you are going, and the insight to know when you have gone too far."

Anonymous

The triad demonstrates to Sam the current state of the system. Debbie and Tom recount other stories that have been completed and tests that have been run.

The Rest of the Story

Debbie begins, "The first features we worked on were 'check-out and check-in' and 'credit card charging'. In the previous chapters, we discussed the detailed tests for the stories associated with these features. We did not talk explicitly about the other stories and tests associated with these features. The acceptance test ideas involved are very similar to the ones that we did talk about."

"To give Sam (and the reader) a full picture, I want to present an overview of some of these other stories and their acceptance tests. Since the system deals with customers and CDs, we need a way to add, update, and delete customers and CDs. We had to add customers and CDs in order to test check-in and check-out. But we didn't go into the stories themselves."

"Adding customers is pretty straight-forward. But you probably want to ensure that a customer doesn't get accidentally added twice. If that happens, than you need a way to merge two customer records into one. You need to have business rules that specify how to determine if you have duplicate customers, such as two records that have the same telephone number, the same credit card number, or the same email address. You also need a way to update or edit an existing customer."

"The other customer story is to delete customers. Maybe this one might be to "deactivate a customer", since a customer may still have outstanding rentals. Or you may wish to keep a record of previous customers so you can welcome them back, or that you don't want them back. Alternatively, you may truly wish to delete a customer, in which case, you need to ensure that all references to that customer are also deleted to maintain what is called referential integrity. That means that all references to that customer in rentals, rental history, or anywhere else also have to be deleted." [IBM01]

Debbie continues, "The same is true for adding, editing, and deleting CDs. You may want to add status information, such as lost or damaged, to each CD, rather than delete the CD record. Tom and I can come up with most of the tests for adding, editing, and deleting items, since they are pretty common. But we need to work with you to come up with the business rule tests such as seeing if two customer records are the same, deactivating customers, and setting CD status."

"As we mentioned in Chapter 9 the acceptance tests we develop for all of these stories can be run through the mid-tier. We can also run them through the user interface to ensure that it is correctly connected to the mid-tier. Talking about the user interface brings up its usability that Tom wants to talk about."

Usability

Tom starts, “We've concentrated on the acceptance tests which demonstrate functionality. As I talked about earlier in Chapter 3, there are also usability tests, exploratory tests, and “ility” tests, such as security and performance.”

“Debbie and I worked with Cary, Harry, and Mary on the usability of the check-out and check-in screens. Harry is color blind, so he couldn’t distinguish that messages displayed in red and green had different meanings. So we added textual indicators to the messages to clarify whether they meant ‘this is a problem’ or ‘this is okay’. Mary doesn’t want to use her glasses to read the screen, so we increased the size of the font. These examples represent issues that we typically find in usability testing.”

Tom continues, “We talked to a couple of your customers to see if the rental contract was readable. The wording that your lawyer approved seemed a bit obfuscated.” Sam interjects, “I think talking about lawyers and obfuscation is redundant.” Tom replies, “Agreed. And so did the customers. So we worked on a simple language contract, which all parties approved.” [ABA01]

Separate State From Response

Tom continues, "We'll talk more about separating display from state in Chapter 14, but since we're talking about the what the user sees, I think it's appropriate to introduce the idea now. We discussed in Chapter 9 separating the form of input from the internal logic. For example, whether the input is from typing, the scan of a bar code, or the reading of an RFID tag should not affect the mid-tier tests. This separation makes for less test maintenance."

"When we developed tests for check-out, we listed the error messages as 'CD Rental Limit Exceeded'. For example:"

Check-out CD		
Enter	**Customer ID**	007
Enter	**CD ID**	CD5
Press	**Submit**	
Check	**Error Message**	CD Rental Limit Exceeded

"Debbie coded the test with a reference to an identifier, such as 'Error Message CD Rental Limit Exceeded'. When Cathy decided what should appear on the screen, we put that into a separate table, as:"

Error Message	
Identifier	**Text**
Error Message CD Rental Limit Exceeded	The customer has exceeded the CD rental limit. Sam has set the limit at 3. Please gently inform the customer of the limit.

"These two tables allow for separation of testing. One test verifies that the system produces the right state. The other test confirms that given the state, the output is was is desired."

Tom continues, "Likewise for the Rental Contract that is printed, as we talked out in Chapter 8, the test verified that the data on the contract was correct. Now we have a separate test for the printed contract itself:"

Given data for a rental contract

Rental Contract			
Customer ID	**Customer Name**	**CD ID**	**CD Title**
007	James	CD2	Beatles Greatest Hits

And this template:

Rental Contract Template
The customer named <Customer Name> with the id <Customer ID>, hereafter referred to as the Renter, has rented the CD identified by <CD ID> with the title "<CD Title>", hereafter referred to as the Rented CD, from Sam's Lawn Mower Repair and CD Rental Store, hereafter referred to as the Rentee. The Renter promises to return the Rented CD to the Rentee within a period of time, hereafter called the Rental Period, not to exceed… blah…blah…blah

When the contract is printed,
Then it should appear as this:

Rental Contract Printout
The customer named James with the id 007, hereafter referred to as the Renter, has rented the CD identified by CD2 with the title "Beatle's Greatest Hits", hereafter referred to as the Rented CD, from Sam's Lawn Mower Repair and CD Rental Store, hereafter referred to as the Rentee. The Renter promises to return the Rented CD to the Rentee within a period of time, hereafter called the Rental Period, not to exceed… blah…blah…blah

The Other Tests

Tom resumes, "I often do extensive performance testing on stories. Just as with the other acceptance tests, I make up a table for the desired behavior. For example, if Sam was expecting to have lots of check-out people other than Cary, Harry, and Mary on the system simultaneously, I would make up a table such as:"

Check-out Performance	
Number of simultaneous check-outs	**Response Time Maximum (seconds)**
1	.1
10	.2
100	.3

Tom continues, "Debbie would take this into account while she is developing the software. However I am much more familiar with the performance tools, so I would develop the tests and run then on the pre-production platform."

"Security is an important area for testing. You need to ensure both physical security of the system and software security. You don't want a customer to go behind the counter and check-in a CD that he really isn't checking in. We have name/password security on the screens, but based on usage patterns,

you may need to change the means and timing for that verification. You could have a logon at the beginning of the day or before every rental. You could software verification or you could have employee cards."

"Since you are keeping credit card information, you need to abide by the PCI Data Security Standard. So we'll need tests to ensure that each of the requirements in that standard is met."

[Security01]

"Security is such a broad issue, that I can't really get into much more detail without filling up the rest of this book. Suffice it to say that I can test a system to see if there are known security issues, but I can't test to make sure that it is absolutely secure. Security is not about letting people do things; it's about making sure they can't do things. It's easier to test the former than the latter."

"The entire team can try exploratory testing on the system. As mentioned in Chapter 3, one way to do this is that each person takes on the role of a different persona. Each performs the operations related to that persona and sees how the system feels to them. Issues may be discovered that do not come out in our pre-determined tests. This form of exploratory testing is closely related to workflow testing that Debbie is going to describe."

Workflow Tests

Debbie starts, "Just because we have tests for each individual story does not mean that the system is fully tested. We need to have a test for an entire workflow. The workflow can correspond to a story map as shown in Chapter 11 or a set of story maps. The workflow test verifies that there are no issues between related stories and that the entire flow is usable. An example of a flow test is in Figure 12.1"

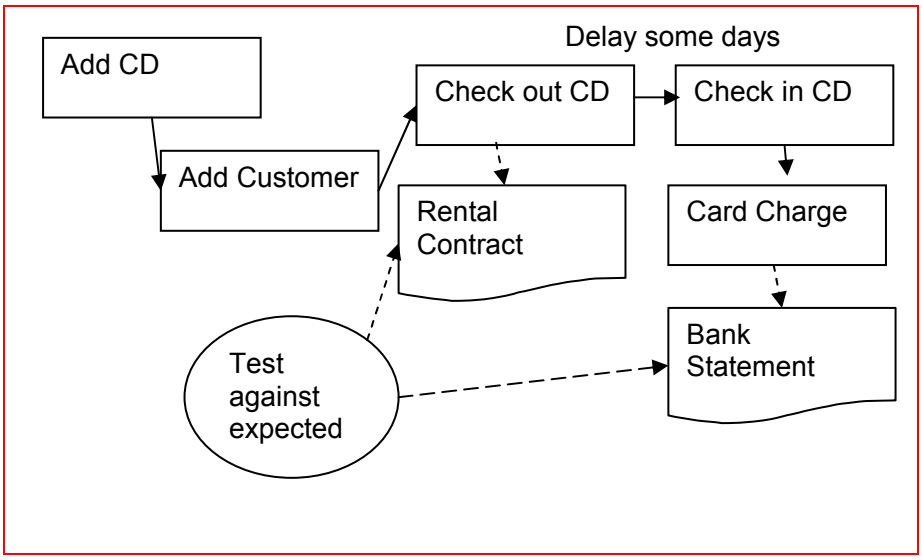


Figure 12.1 *Workflow Test*

Debbie continues, "If the workflow was really complicated, we might have multiple workflow tests that try alternative paths. Workflow tests should be performed on as many platforms as possible to catch issues as early as possible."

Unit Tests Are Not Enough

Several companies make highly available disk storage systems. As part of the system, there is a monitoring module that checks to see if each disk has any problems. (See Figure 12.2) One measure it uses is the response time for a disk. If a disk does not return requested data in a certain amount of time, such as one second, then the monitor reports a failure. Tests were run to ensure that the monitor operated properly.

Much later a second requirement was added to make the system green. To save power, a power saver monitor turns a disk off if it is not accessed for a certain amount of time. The tests for that requirement also passed.

Testing the individual pieces is insufficient to prove the entire system. When the new feature was tested in an integrated environment, the disk monitor would signal failures at random times. The power saver was turning off disks when they had not been accessed in the recent past. The next time a powered-down disk was accessed, it took more than a second to respond, since it had to be powered up.

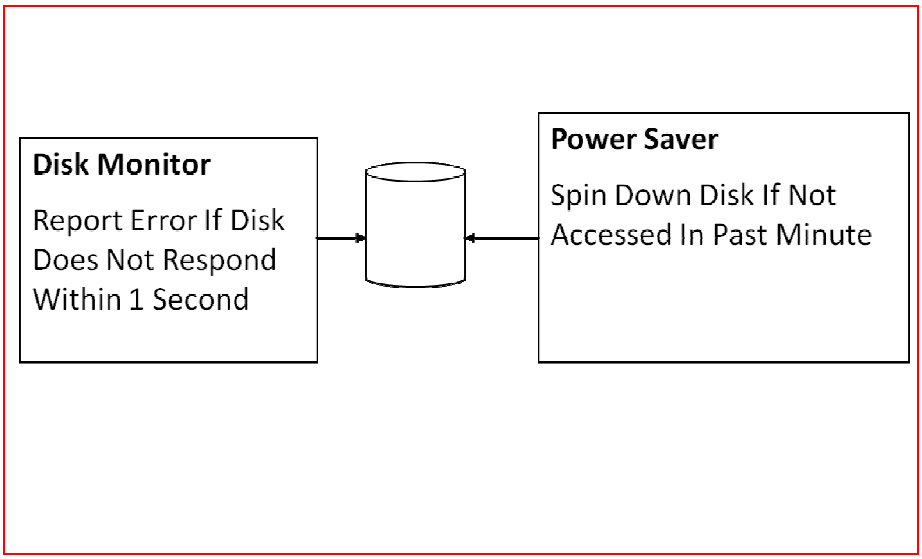


Figure 12.2 *Cross Story Interaction*

The Stakeholder's View

Sam speaks up, "Cathy has been keeping me in the loop about your discussions. So let me ask the bottom line question - how quickly can you get the system into operation? Other than the stuff you've already outlined, what else do you need to do?"

Debbie replies, "We already have a way to get customers and CDs into the system. We don't have all the data. If we had a spreadsheet with all the data, then we could input it into the system."

Sam states, "I'll have Cary, Harry, and Mary start inputting all that information. What else?"

Debbie answers, “You’ll need a transition plan to go from the old system to the new one. You could start with doing rentals both ways for a little while, or doing all new rentals in the new system, or doing rentals in the new system for just a few customers.”

Sam says, “Let’s work on that in a little while. Is that it?”

Debbie replies, “Anything else you can think of Tom?” Tom responds, “I think that’s it.”

Sam states, “Then let’s go for it. The sooner this starts to happen, the quicker we can start renting more CDs with the same number of people and thus make the money to pay you for all your hard work.”

Cathy injects, “I think it’s time to look back at the charter and its objectives. The one we’re dealing with right now is ‘Within two months, CD check-outs and returns will be processed in 50% less time.’ From what I’ve seen so far, it looks like that will be met. But as you say, the sooner we get it working, the sooner we’ll find out.”

Where We're Going

We've followed the triad, Cathy, Debbie, and Tom; Sam the sponsor; and the users - Cary, Harry, and Mary from the initial project charter through the first deliverable. Along the way, you've seen acceptance criteria developed for the project and the high level features. You've seen how acceptance tests are created for the detail of stories as expressed in use cases. The entire flow of the process has been explored. In the next part, we'll cover more details for aspects of the flow.

Summary

- It is insufficient to have just acceptance tests that revolve around functionality of a system
- Acceptance criteria need to be established for usability, security, performance, and other “itlities”
- Workflow tests in pre-production environments catch inter-story issues
- Developing in deployable chunks allows for quicker cost recovery

-2

Part Two - Details

This part explores details of acceptance testing. It starts with separating concerns to make things easier to test and maintain.

Chapter 13

Separate to Simplify

"Life is like an onion: you peel it off one layer at a time, and sometimes you weep"

Carl Sandburg

The triad discusses a new story from Sam – letting people reserve CDs on line. The story illustrates how separation of issues makes simpler tests.

Reservation Story

Cathy starts off, "Sam has an idea for a website. It sounds like a separate project. The web site connects back to the charter, since we want to provide additional services to our customers. He has come up with two ideas. He wants to allow customers to reserve CDs on the web. He also wants them to be able to search CDs. We haven't worked out the details on the second story. But he's already decided on the first."

"Sam has worked out a pretty elaborate business rule for whether a customer should be allowed to reserve a CD. He created a table, since he's been following along with our discussions. The table looks like the one that follows."

Debbie interrupts, "That table looks pretty complicated, but I've seen some business rules that look like it. The author suggests that the readers shouldn't get hung up on the details."

Cathy replies, "I think Sam can be a little complex sometimes. He wants to base the decision on allowing someone to reserve on a number of criteria. They include the number of times they rented in the past month and the cumulative number of rentals since they have been a customer and the number of late returns for the past month and since they began renting. Sam also has a few people who are his favorites, whom he wants to be allowed to reserve unless they have a really bad rental history."

Allowed To Reserve Business Rule			
Monthly Rentals	**Cumulative Rentals**	**Sam's Favorite Customer**	**Allowed To Reserve?**
If Rentals Past Month > 30 and Late Rentals Past Month <= 1	If Cumulative Rentals > 100 and Late Cumulative Rentals <= 2	Does Not Apply	Yes
Does not apply	Does Not Apply	Yes	Yes
If Rentals Past Month > 30 and Late Rentals Past Month <= 3	If Cumulative Rentals > 300 and Late Cumulative Rentals < 10	No	Yes
If Rentals Past Month > 20 and Late Rentals	If Cumulative Rentals > 200 and Late Cumulative Rentals < 5	Unknown	Yes

Past Month <= 3			
Does Not Apply	Anything Else	Does Not Apply	No

Cathy asks, “So what do we do with this?”

Tom replies, “With all these comparisons and complex conditions, this is a hard table to understand. We can break this table up into smaller tables, if Sam lets us or you let us, in lieu of Sam being here. As David Parnas states [Parnas01] , tables can clarify the requirements. And smaller tables can add more clarification ”

Cathy responds, “It’s as clear as mud. So let’s break it up.”

Tom starts, “Let’s start with the Monthly Rental column. We can separate out the values into separate fields and put the comparisons into single cells. I like to make up names for the result of each comparison. My suggestion is to call the results ‘Monthly Rental Levels’ or ‘MRLevels’ to keep it short. If there were meaningful names we assign to each result, we might name them ‘MRExcellent’ or ‘MRGood’ and so forth. But in this case, let’s just label them with letters. The table for Monthly Rental Levels looks like:”

Monthly Rentals Levels Calculation		
Rentals Past Month	**Late Rentals Past Month**	**Monthly Rental Level**
>30	<= 1	MRLevelA
>30	<= 3	MRLevelB
>20	<= 3	MRLevelC

Tom continues, “Now here are some tests for these Monthly Rental Levels. As you can see, there are plenty of tests for just this one little piece.”

Monthly Rental Levels Tests		
Rentals Past Month	**Late Rentals Past Month**	**Monthly Rental Level**
30	1	MRLevelA
30	2	MRLevelB
31	1	MRLevelA
31	2	MRLevelB
31	3	MRLevelB
32	4	??
20	3	??
21	3	MRLevelC

Tom continues, “By breaking down the original table into smaller tables, we can see whether we have left out anything. Creating some tests for just the Monthly Rental rule shows that there are possibilities that have not been covered. Perhaps these possibilities may never occur during production. But at least we’ve identified them and Debbie can be sure to make allowance for them in the implementation. She could at least record that they occurred or put up a dialog box or do whatever is appropriate."

Debbie interjects, "It seems like there should be a MRLevelD as the default level if none of the conditions are met. That will make it easier to keep track of those possibilities when they occur. ”

"We can do the same thing for the Cumulative Rentals. I'll leave that as an exercise to the readers, as its practically the same as for Monthly Rentals. But here's a start:"

Cumulative Rentals Levels Rule	
Condition	**Level**
If Cumulative Rentals > 100 and Late Cumulative Rentals <= 2	CRLevelA
If Cumulative Rentals > 300 and Late Cumulative Rentals < 10	CRLevelB
If Cumulative Rentals > 200 and Late Cumulative Rentals < 5	CRLevelC
Anything Else	CRLevelD

Tom continues, "With these tables taking care of the details for Monthly Rentals and Cumulative Rentals, our table looks like:"

Monthly Rentals	**Cumulative Rentals**	**Sam's Favorite Customer**	**Allowed To Reserve?**
MRLevelA	CRLevelA	Does Not Apply	Yes
Does Not Apply	Does Not Apply	Yes	Yes
MRLevelB	CRLevelB	No	Yes
MRLevelC	CRLevelC	Unknown	Yes
Does Not Apply	CRLevelD	Does Not Apply	No

Tom continues, "Have we covered all of the cases? No. But at least it's clearer now what all the cases are. There are several combinations of MRLevels, CRLevels, and Sam's Favorite Customer that do not appear in the table. Here are a few:"

Monthly Rentals	**Cumulative Rentals**	**Sam's Favorite Customer**	**Allowed To Reserve?**
MRLevelA	CRLevelB	Yes	No??
MRLevelA	CRLevelB	No	No??
MRLevelA	CRLevelB	Unknown	No?

"We need to ask Sam whether the answer is 'yes' or 'no' in these cases. He could simply say that the answer is no in all other cases. That would make Debbie's life easier, as well as mine. The point is that separating things into separate tables decreases the amount of information that needs to be absorbed and makes the tests cleaner."

Rental History

Debbie starts, "Now to allow reservations based on Sam's business rule, we need to keep track of the rentals for each customer. We could keep separate information on each rental. At this point, we don't have any need for the history. We just need the count of rentals for the month and total number of rentals. So for each customer, we might have:"

Given this rental history:

Customer Data						
Customer ID	**Name**	**Rentals Past Month**	**Late Rentals Past Month**	**Cumulative Rentals**	**Late Cumulative Rentals**	**Sam's Favorite**
007	James	100	3	300	30	Yes
86	Maxwell	200	1	400	30	No

Then determine if reservation is allowed:

Reservation Allowed	
Customer ID	**Allowed?**
007	Yes
86	No

Tom continues, "When I test the overall system including the user interface, I'll look up both 007 and 86 and see whether the reservation is allowed or not. If we had a system in production whose data was available on the pre-production system, I would try to find two customers - one who was allowed to reserve according to the rule and one who was not. It would require a little bit of searching, but I'd be able to do the test against real data."

Debbie resumes, "We need to test that the system calculates rental history correctly So here's a test:"

Given this rental history

Rental History Data	Customer ID = 86	
CD ID	**Rental Due**	**Rental Returned**
CD3	1/1/2010	1/1/2010
CD5	1/3/2010	1/3/2010
CD7	1/3/2010	1/4/2010
CD2	2/11/2010	2/12/2010
CD4	2/13/2010	2/13/2010
CD6	2/13/2010	2/14/2010
CD7	2/14/2010	2/14/2010

And the

Test Date
Date
3/1/2010

Then the monthly and cumulative rental counts should be:

Rental Summary					
Customer ID	**Name**	**Rentals In Past Month?**	**Late Rentals In Past Month?**	**Cumulative Rentals?**	**Late Cumulative Rentals?**
86	Maxwell	4	2	7	3

Debbie continues, "The test does not imply whether we actually keep a rental history or not. If we do, we might use the history for other features that are scheduled to be developed very soon. All we need now is to change the numbers whenever a check-in occurs. When the check-in happens, the number of rentals would be incremented by one. If the rental was late, the number of late rentals would also be incremented by one. Once a month, the system would clear out the rental count for the past month."

"It might be easier for me to keep a history. The check-in would add the data for a rental to the history. The Rental Summary calculation would find all the rentals for a customer and calculate the counts. In either case, the test is independent of the way I perform the calculation."

Tom concludes, "Now that we've shown how to simplify a business rule by separating it, we'll continue in the next chapter with how to simplify by separating the user interface from the business logic."

Summary

- Simplify business rules by separating them in component parts

- Create tests for the component parts
- Use the simpler components to determine missing logic

14

Separate Display From Model

"To each his own. (Suum Cuique)"

Cicero

Debbie explains how to make more maintainable tests by separating what appears to the user from the logic in the underlying business model.

Decoupling the User Interface

Debbie starts off, "Cathy, Tom and I like to separate the state of the system from how that state is displayed. For example, in the CD reservation story, we created tests to make sure that we determine correctly if a customer is allowed to reserve. As Figure 13.1 shows, we have reservation tests. We have not talked about how that condition is displayed and made up tests for the user interface."

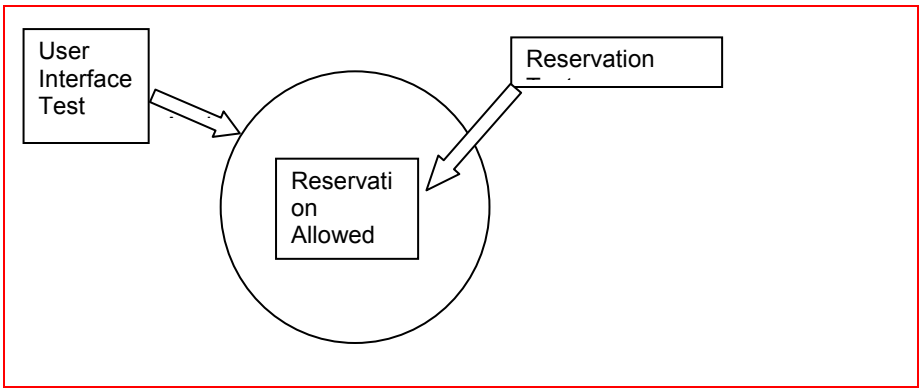


Figure 13.1 *User Interface and Logic Tests*

Cathy continues, "You have at least three ways that we could show how a customer could or could not reserve a CD. If they are not allowed to reserve, you could hide the reserve option. You could disable the reserve option. Or you could have the reserve option go to a dialog box that informed the customer that they were not allowed to reserve. Here are three ways displayed in tables:"

Display For Reservation Allowed	
Allowed	**Reserve Button Displayed?**
Yes	Yes
No	No

Or

Display For Reservation Allowed	
Allowed	**Reserve Button Enabled?**
Yes	Yes
No	No

Or

Display For Reservation Allowed	
Allowed	**Reserve Button Goes To?**
Yes	Make Reservation Dialog
No	Sorry No Reservation Dialog

Debbie continues, “Let’s take a look at how these variations might be described in specific tests.”

Given that these customers are either allowed or disallowed to reserve

Reservation Allowed		
Customer ID	**Allowed**	**Name**
007	Yes	James
86	No	Maxwell

Then display reserve button if customer allowed to reserve

Customer Screen	
Customer	Maxwell
	No button

Customer	James
	Reserve

Or
Then enable reserve button if allowed to reserve

Customer	Maxwell
	Reserve –disabled

Customer	James
	Reserve - enabled

Or
Then display reservation allowed to reserve and put up a different dialog box

Customer	Maxwell
	Reserve

On click on Reserve for Maxwell, display

No Reservation Dialog
Sorry Maxwell, you cannot reserve CDs

Customer	James
	Reserve

On click on Reserve for James, display

Reservation Dialog	
James, What CD would you like to reserve?	
CD Title	Beatles Greatest Hits
	Submit reservation

Tom interrupts, "Now when Sam or you change your minds about how the ability to reserve should be displayed, all Debbie and I have to do is to change the test to one of these. These tests do not have to be automatic tests. I can manually run this test. If there gets to be too many of these manual tests, then I could use a user interface test automation tool. I would also use the tool if I want to automatically verify the user interface when the application is being built, as shown in Chapter 3."

Tom continues, "Suppose we had not separated out the display from the logic, such as:"

Show Reserve Button			
Monthly Rentals	**Cumulative Rentals**	**Sam's Favorite Customer**	**Show Allowed to Reserve?**
MRLevelA	CRLevelA	Does Not Apply	Reserve Button Enabled
Does Not Apply	Does Not Apply	Yes	Reserve Button Enabled
MRLevelB	CRLevelB	No	Reserve Button Enabled
MRLevelC	CRLevelC	Unknown	Reserve Button Enabled
Does Not Apply	CRLevelD	Does Not Apply	Reserve Button Disabled

Tom continues, "Whenever you or Sam made a change in how to display the allowable reservation, I would have had to change this bigger table. I'm sure I would make a mistake at some point. This separation of state from display translates both into easier to test and simpler code. From a testing standpoint, I just have to confirm that the user interface displays the correct result in two cases – for 007 and 86. With this table, I have to see that the user interface displays the correct result in five cases – one for each row in the table. "

A Little Internal Design

Tom continues, “We haven’t talked much about the internal design of an application. In our initial description, I suggested how having a test for a business rule forces Debbie to make the module for that business rule easily available as shown in Figure 13.1”

“There is a design guideline called Model-View-Controller. Now some people call it a design pattern, others call it a application framework. If you’re interested, I can go into the reasons why there are these other terms.”

Cathy asks, “Does it really make a difference?”

Tom replies, “From your standpoint, not really. So I’ll just state the practice of this guideline. You separate the model of something from the view and controller. In this case, the model is your business rule for determining whether a customer can reserve a CD. The view is how you see the reservation allowed decision – button or dialog box. The controller is how you can make a reservation – the reservation dialog box. The view and the controller should be coded separated from the model. I’m not going to get much into code. The *Prefactoring* book suggests that you the customer should be able to read the code that Debbie writes. So Debbie, will you show Cathy how you might write it?”

Debbie answers, “I haven’t written the code yet, but this is close to how it might look. Suppose that the method that determines whether a customer is allowed to reserve is called:

```
Boolean Customer.allowedToReserve()
```

“Boolean means that this method returns either true or false. True and false is equivalent to 'yes' and 'no'. What this method does is obvious from its name. It returns whether a customer is allowed to reserve or not by calculating the result according to Sam's business rule. The way the display code would look if the button was to be enabled or disabled would be:”

```
if (customer.allowedToReserve())
    enableReserveButton()
else
    disableReserveButton()
```

“If the button was to be shown or hidden, it might look like: “

```
if (customer.allowedToReserve()
    showReserveButton()
else
    hideReserveButton()
```

“And finally, if you wanted to display different dialogs, it could be:”

```
OnReserveButtonClick()
    {
    if (customer.allowedToReserve())
        displayReservationDialog();
    else
       displayNotAllowedToReserveDialog();
    }
```

Debbie continues, “The actual code will be slightly different, depending on whether the application is standalone or on the web. But it’s pretty close.”

Summary

- Decouple the user interface from the business logic
- User interface code should use mid-tier logic code

Chapter 15

Events, Responses, States

"I just dropped in to see what condition my condition is in"

Mickey Newbury, sung by Kenny Rogers of the First Edition

Debbie describes to Cathy the event-response table way of capturing requirements. Tom explains how the state transitions caused by events are tested.

Event Table

Debbie explains, "The CD rental process is mostly driven by user actions. So employing use cases was a natural fit for eliciting requirements. There are other ways that we could discover requirements. A popular technique is the event table. You define events that occur and determine how the system should respond. An event could be something a user initiated or something that a piece of hardware signaled. It could also be a particular time, such as the first of the month, or a time period, such as every hour. The response of the system could be a visible output or a change in the internal state."

"Let's give it a try for the Sam's system to see if we've missed anything. I'll give a few examples to get the ball rolling. We'll start with just the events and then fill in the responses later"

Events for CD Rental	
Event	**Notes**
Customer rents CD	Human initiated
Customer returns CD	Human initiated
First of the month	Specific time
Rental period ends	Period of time
Bank statement arrives	External event
Customer enters store	Human initiated
Chicken Little announces the sky is falling	External event

Debbie continues, "I gave the last two examples of events that might occur during a brainstorming session. If you are really brainstorming, you shouldn't filter your ideas. Just write them down. After the first part of the session is over, then the group can determine whether the ideas are in or out of scope. Now that I've given some ideas, Cathy, we should see what other events we can come up with."

The triad brainstorms for a while and develops a whole set of events. A few of these are:

Events for CD Rental	
Event	**Notes**

Customer reports CD is lost	Human initiated
Inventory Maintainer cannot find CD	External event
Counter Clerk drops CD and it breaks	External event

Debbie continues, "Now we need to come up with how the system should respond to all of these events. Cathy, you'll be the point person on this. Tom and I will chime in as necessary."

After a period of time, the triad winds up with the following:

Events for CD Rental		
Event	**Response**	**Notes**
Customer rents CD	Record CD as rented Print rental contract	Human initiated
Customer returns CD	Record CD as returned Charge for rental	Human initiated
First of the month	Print inventory report	Specific time
Rental period ends	Notify customer of end of rental	Period of time
Bank statement arrives	Nothing	External event
Customer enters store	Nothing	Human initiated
Chicken Little announces the sky is falling	Nothing	External event
Customer reports CD is lost	Record as lost Charge for CD	Human initiated
Inventory Maintainer cannot find CD	Record as lost	External event
Counter Clerk drops CD and it breaks	Record as broken	External event
Counter Clerk sees CD is dirty	Set aside to clean	External event

Debbie finishes, "Anything that does not require a response from the system is out of scope. For example, the bank statement arriving does not have a response. Most of the events that are human initiated will get turned into a use case. Some of the external events, such as dropping a CD, may also get turned into a simple story. Making up a use case might be overkill, since there really are not many details associated with them."

Tom chimes in, "We need to create a test for every one of the events, even the simple ones. The tests clarify exactly what the system response should be. For example, dropping a CD might have something like;"

Dropping a CD

Given a CD in the inventory

CD		
ID	**Status**	**Rented**

CD5	Okay	No

When it is broken by the Counter Clerk or Inventory Maintainer, record the event

Record Broken CD		
Enter	**CD ID**	CD5
Press	**Submit**	

Then the CD status should change

CD		
ID	**Status**	**Rented**
CD5	Broken	No

Tom continues, "We could make up tests for all the conditions. Suppose the CD was rented and it was returned broken. If you don't want the status to be very detailed, we could group all losses into a small number of categories, such as things that are irreversible, as a broken CD, and things that are possibly reversible, just as a dirty CD or a missing one. After all a missing one might be found sometime. "

States

Tom resumes, "Entities such as CDs take on different states or conditions. The CD transitions from one state due to an event, such as the ones shown in the event-response table. Documenting the states and the transitions is a collaborative effort, just as for the event-response table. The outcome of the effort is a list of the states and a map of the transitions."

Based on the events, more discussion, and some simplification, the triad agrees on the following states:

CD States	
State	**Meaning**
Ready To Rent	In inventory, ready for renting
Rented	Customer has it on rental
Irreversible Loss	e.g. broken, totally scratched
Reversible Loss	e.g.. dirty, cracked case
Missing	Not rented, but not found in inventory
Lost	Customer reports CD is lost

Tom continues, "We can document the transitions in two ways. The first is a state diagram. With this diagram in Figure 15.1, the states are shown in circles, and transitions are shown as lines with labels. With each state, you may have associated data. For example, for the rented state, you have the date the CD was rented and the customer it was rented to. The black circle on the left is called the initial state. It points to which state is the first one. The small black circle on the right is the terminal state. Any state

can transition into it. When the terminal state is reached, the state does not exist any more." Debbie interjects, "Arnold Schwarzenegger, the Terminator, got to it."

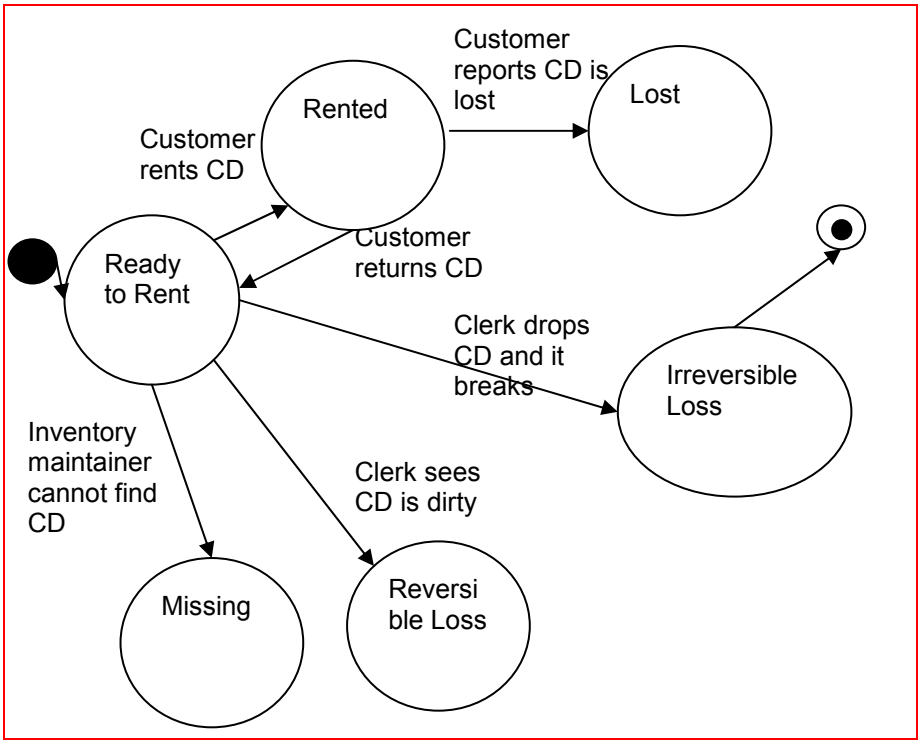


Figure 15.1 *State Diagram*

Tom resumes, "You can specify the states and events in a table. Initially this may display the same information as parts of the event/response table. However it concentrates on a single entity, the CD, rather than all the the entities in the system."

CD State						
States / Events	**Customer Rents CD**	**Customer returns CD**	**Inventory maintainer cannot find CD**	**Counter Clerk drops CD and it breaks**	**Customer reports CD is lost**	**Counter Clerk sees CD is dirty**
Ready To Rent	Rented		Missing	Irreversible Loss		Reversible Loss
Rented		Ready To Rent			Lost	
Irreversible Loss						
Reversible Loss						
Missing						
Lost						

"We see that there are states that are transitioned into, but not out of. If that is the case, then they are terminal states. Otherwise, we need to identify the event which takes the CD out of those states." After some discussion, the triad comes with these additional events. The entire table is not repeated, since it would exceed the width of the page.

CD State				
States / Events	**Customer reports CD is found**	**CD prepared for rental (cleaned)**	**Create monthly inventory report**	**CD is found**
Ready To Rent				
Rented				
Irreversible Loss			Terminal Remove CD from system	
Reversible Loss		Ready to Rent		
Missing				Ready to Rent
Lost	Rented			

Tom resumes, "With the state-event table, we can easily see the blank cells. A blank cell represents that an event should not occur for a particular state or if it does, it should not cause the state to change. You should examine all blank cells to ensure that you have covered all the bases - that is, all possible state transitions due to events. You might want to fill in a blank cell you have examined with some indicator, such as 'N/A' for 'not applicable' to show that you have thought about that state/event combination."

"While the primary purpose of the state table is to show transitions, you can put other responses and state data such as the date rented into this state table as well. For example, I showed the action of 'Remove CD from system' as an action for the state/event combination of Irreversible Loss/Create Monthly Inventory report?"

"We should have a test for every state transition. Some of the transitions are already covered by other tests. The ones for check-out and check-in already cover the transitions between 'Ready to Rent' and 'Rented'. Other transitions may show that we may have additional user stories that need to be implemented, such as a "Prepare CD For Rental". The tests for those stories would show the transition from 'Reversible Loss' to 'Ready to Rent'.

Events, State, Response

Debbie starts off. "Cathy, I'd like to give another example of event, state, and response. We talked about how every external event causes a system to either produce an externally visible response or to change its internal state (e.g. persistent data) or both. An internal state change causes the alteration of an externally visible response in the future. For example, let's say you are keeping the addresses of your customers. Now you change the address of a customer as shown in Figure 15.2

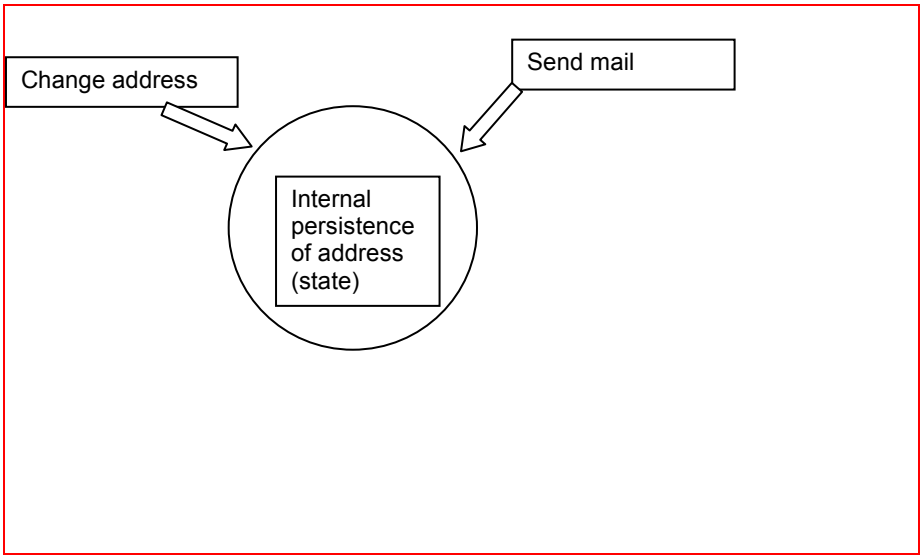


Figure 15.2 *Internal State*

Event/Response	
Event	Address Changed
Response	Address Change Confirmation
Internal State Change	Address Updated

"In the future, when you send mail, the new address is used."

Event/Response	
Event	Send mail
Response	Send to current address of customer

"An acceptance test for an internal state change could confirm that the state has been changed. Or it could be combined with another test that shows the result of that changed state. So the Change Address – Send Mail test would flow together."

"Alternatively, if the address is kept in a repository that is external to the system, then the update is part of the response."

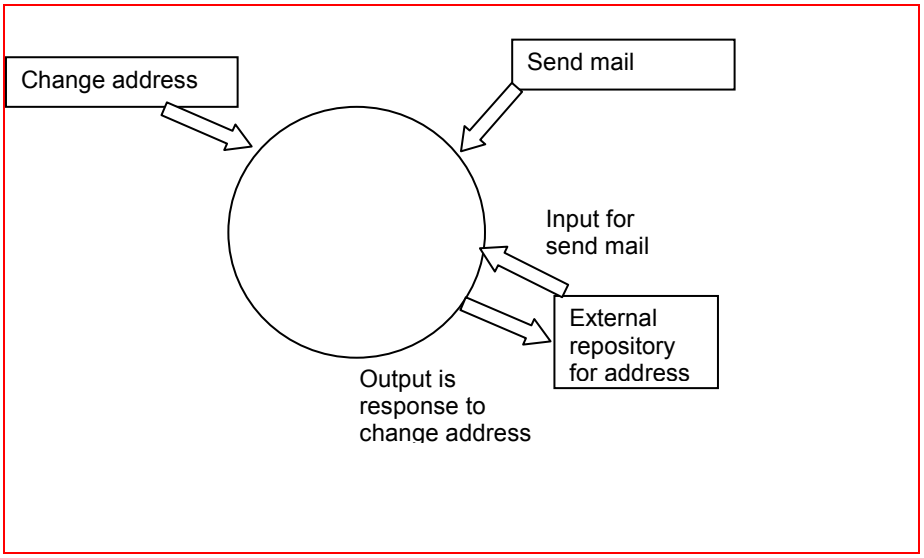


Figure 15.3 *External State*

Event/Response	
Event	Address Changed
Response	Address Change Confirmation
Output	New Address output to external repository
Internal State Change	None

"In this case, the output is expected. The acceptance test can verify this output to another system, just like we did with the credit charge in Chapter 11. In creating a response to an event, a system may use data that comes from the external world. That data may affect the response. In this case, the new data in the repository causes the response to be changed. In this case, the developers may create a test double to simulate the changed data in the external repository. That will be detailed more in Chapter 20.

Event/Response	
Event	Send mail
Response	Input address from external repository
	Send mail to that address

Transient or Persistent

Debbie continues, "Just to be complete, a state change may be transient or persistent. A transient state change exists for a short period of time. For example, you could change your address just for a single order or for all the orders made during a web session. A persistent change would change your address permanently or at least until you changed it again."

"The order itself could be transient or persistent. Suppose you have not finished an order before disconnecting from a web session. A transient incomplete order would go away and not be there when you reconnect. A persistent incomplete order would come back when you log back on."

A Question

Debbie finishes with a Zen-style question on states, "By the way, if the internal state changes, but it never affects anything directly or indirectly seen in the outside world, is that state change necessary? Does a tree make a sound when it falls with no one around?"

Summary

- Event / response tables are a complementary way of eliciting requirements.
- Every event/response combination should be covered by an acceptance test
- A state table documents the events that cause the state of an entity to change.
- Every state/event combination should be covered by an acceptance tests
- State tables and event/response tables may show the information, but organized with a different emphasis.

Chapter 16

Developer Acceptance Tests

"What is good for the goose is good for the gander."

Anonymous

The triad has developed acceptance tests for stories that Cathy created. When a developer needs a component from another developer, the requestor should create acceptance tests for that component.

What? Me Too?

Tom starts off, "Acceptance tests are not generated by just the customer unit. Many software development organizations have groups that develop common modules or components for other groups. The requesting group needs to supply acceptance tests for those modules."

"For example, often specialized developers create custom user interface components. These components are used in web pages by the user interface developers. Let's take a look at a couple of common components. One is a text box that accepts a formatted string, such as a phone number or an email address. The other is a tabular style component that allows for sorting by the values in columns. Debbie can explain the tests since she's the one who created them. As you'll see, the tests look like specifications or vice versa."

Debbie begins, "A customer wants a field on the display where a phone number can be entered. Obviously the data entered in the field should be checked for validity. The format differs between nations, so to keep it simple, I show just a U.S. phone number. The 'N' in the table means a number or digit."

Formatted Field			
Field Type	**Example**	**Format**	**Validation**
U.S. Phone Number	1-919-555-1212	1-NNN-NNN-NNNN	Area code must be valid

Tom interrupts, "Here are some of the tests that I would create for the U.S. Phone Number. By now, the reader should know that the tests not shown are left for an exercise."

U.S. Phone Number Component Tests		
Value	**Valid?**	**Notes**
1-919-555-1212	Y	
1-000-555-1212	N	Bad Area Code

Debbie continues, "Now suppose Don is the developer who is going to create this component. The first thing I need to do is give an acceptance test for what it should look like. For example, will it display the dashes immediately or as a person types the digits? Let's suppose the former. So I create a image of what the component should look like in Figure 16.1. The image is part of the acceptance test. It's just one that has to be verified manually. The color and font also need to be checked."[8]

(*** Make into screen shot image ***)

Phone Number	1- - -

Figure 16.1 *State Diagram*

Debbie continues, "I need to specify to Don what should occur if the phone number is incorrectly formatted. Should there be a beep? Should a message be immediately displayed? Should a message be added to a message list and displayed when the submit button for the entire form is pressed? Since the phone number box will be used repeatedly on every form that requires one, this decision may need to be made by a corporate user interface standards group. Let's assume that the error should be displayed when the user finishes an entry in the field. So I create a table that describes how the error should appear."

U.S. Phone Number Errors		
Error	**Display Message?**	**Notes**
Bad Area Code	"You entered an invalid area code"	All errors should be displayed to the right of entry field on leaving the field

"I need to show Don what I want the component to output when the value is submitted. This table shows that the dashes should be included in the output value. Another possibility is that the dashes should be eliminated."

U.S. Phone Number Output		
Field ID	**Field Entry**	**Output?**
"HomePhone"	1-919-555-1212	"HomePhone=1-919-555-1212"

Debbie continues, "Don makes up a form with a phone number field and a submit button. He also makes up a form with three entry fields – one prior to the phone number field and one after it – and a submit button. He then demonstrates to me that the component passes the tests for both a valid and an invalid entry. The two different forms show that the text field works all by itself, and that it works with other entry fields. The final test shows that the field sends the correct information when the submit button is pressed."

"This component is a user interface component, so there will be some manual tests to ensure that it is acceptable to a human being. Automated tests can be created to check that the component continues to work even if changes are made in its underlying implementation. "

Tabular Display

[8] It's possible to automatically verify that two images match, but differences that are imperceptible to the eye may cause the automatic match to fail

Debbie resumes, "Let's look at another component which is more complicated. You often need to display a table of items. You may want the user to sort the table by the values in each column or have links within the table. You also need to determine what to do if there are no values in the data to be shown in the table and what to do if the number of values is large."

"Let's start with some of the potential inputs to a table display. It doesn't matter whether these values come from a program component, such as a java class, or a database query. That's a matter of connecting up the table to the appropriate code. The first input is a simple set of data:"

Table Data		
Name	**Date**	**Amount**
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00

"I need to show Don how this data should appear on the screen. Suppose we wanted to have shaded column heads and shading to appear on every other row. We show this as the desired output. Note that this test is going to be manual. We don't want to specify how the shading occurs – by Hypertext Markup Language (HTML) attributes or Cascading Style Sheets (CSS) styles or some other set of acronyms. We are only concerned with the outward appearance." (*** Shading in table below ***)

Name	**Date**	**Amount**
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00

"Now we need to clarify the contract for the table display component. Should it be the my job to ensure that the component never gets passed an empty table or a really big table or should it be the component's responsibility to handle these cases. If the latter, then we need to create some tests for those two conditions:"

Given the following data:

Table Data		
Name	**Date**	**Amount**

Then the following should appear on the screen:

Name	**Date**	**Amount**
No matching records		

"And then I need to create tests for a larger number. Suppose the table should only display 20 rows at a time. I would have a test that shows what should happen if there were 25 rows to display. "

Given the following data:

Table Data		

Name	Date	Amount
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00
Plus 22 more rows…		

Then the display should look like:

Name	Date	Amount
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00
Plus 17 more rows…		
Previous 20 Next 20 1-20 of 25		

(*** 'Previous 20' should be in light grey ***)

Tom interrupts, "As a tester, I might come up with a few more tests. For example, I would like to see what happens if there are exactly 20 or exactly 40 or some large number such as 10000000."

Debbie speaks up, "When Tom comes up with that last one, Don would throw something at him. I and Don might then agree that the component does not have to handle more than a reasonable number of components, such as 1000. That's an example of how coming up with tests in advance helps to hone in on the requirements."

Tom replies, "That's my job – to get things thrown at me, rather than the program getting thrown a loop by these situations."

Debbie continues, "Instead of the buttons at the bottom to see the other items, the table could display 20 items, but have a scroll bar if there are more. So I'd have a test that provided 1000 items of data to see how fast the component was able to scroll."

"Now the table data might have links associated with them. Suppose that clicking on the name was supposed to go to a particular page associated with the data. We would include the link in the supplied data:

Given the following data:

Table Data			
Name	**Date**	**Amount**	**Link**
James	2/1/2010	$5.00	James.html
Maxwell	1/3/2010	$.01	Maxwell.html
Agent	12/12/2010	$10.00	Agent.html

Then the display should appear as:

Name	Date	Amount
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00

And the clicking on the following should go to the indicated page

Click On	
Name	**Go to page?**
James	James.html
Maxwell	Maxwell.html
Agent	Agent.html

"Now the last thing is sorting. When the user clicks on a column header, the rows should be sorted by the values in that column. To show what I mean, I give a few tests to Don. Now I could give an example that has more rows than fit on the screen to show that sorting is on the entire set of data, not just the rows displayed on the screen. But let's keep it short. Here's a test for the date column."

Given the following display

Name	Date	Amount
James	2/1/2010	$5.00
Maxwell	1/3/2010	$.01
Agent	12/12/2010	$10.00

When the user clicks on the Date column header, the rows should be sorted by date, with the earliest date first.

Name	Date	Amount
Maxwell	1/3/2010	$.01
James	2/1/2010	$5.00
Agent	12/12/2010	$10.00

"Now you have a lot of other considerations in the sorting. For example, what if there are two values that match. Which should come first? Or if you click a second time on a column header, should it reverse the sort? The tests for those details are left as an exercise for the reader."

Summary

- Developers should supply acceptance tests for display components and modules created by others
- Acceptance tests for display components should include combinations of inputs that result in different display appearance or response
- Acceptance tests for user interface components may not necessarily be automated

Chapter 17

Decouple with Interfaces

Wizard of Oz: "Pay no attention to that man behind the curtain"

The Wizard of Oz

Developers can create acceptance tests for service implementations. The tests can be both for accuracy and for performance.

Service Provider

Debbie needs a service such as verification of a ZIP code for an address. She is not going to code the service herself, but obtain it from another group. Dave is the developer who is going to provide that service. As a backup plan in case Dave cannot implement the service, she may purchase it from an external vendor.

As the requestor, she needs to create acceptance tests for the service that Dave will use to test his implementation. The tests should be independent of the service provider. Ideally she should not have to rewrite any of her production code in the event that the service provider is changed. So she creates an Application Programming Interface (API) that Dave and other service providers must implement. [Pugh01] The tests that she writes as well as her production application are coded to that interface.

The Interface

Debbie needs to verify that the ZIP code is correct for an address. She creates an interface that describes what she is after:

```
interface ZipCodeLookup
     {
     ZIPCode lookupZipCode( String streetAddress,
              String line2Address, String City, State state) throws
              ZIPCodeNotFound
     }
```

"ZIPCode" is a data class that contains the ZIP Code. The lookupZipCode method needs to indicate an error if the ZIP Code cannot be found. From a programmer's point of view, if the ZIP Code was not findable, the method could return the value of null or throw an exception that contained further information. The "State" is a data class that contains a reference to a valid U.S. state or territory.

There are other things that she may need, such as the ability to get the city and state that correspond to a ZIP code. If so, she would add that to the interface.

Debbie provides the following table as an example of the tests she is going to run against the interface.

Zip Code Lookup	

Street Address	Line2 Address	City	State	ZipCode?
1 East Oak Street		Chicago	IL	60611
101 Penny Lane		Danville	VT	05828
1 No Place		Nowhere	OK	NotFound

Dave may create an implementation that uses a subscription to the United States Postal Service web-service, a corporately-owned address validation program, or some other system. If Debbie needed a quick way to do an end-to-end test, he might create a browser-simulator version that interacts with the usps.com web site and provides the answers found on that site. If Debbie need a way to have a quick unit test for her code, she might create a "test double" (see Chapter 11) for USPSZipCodeLookup. The test double would only return a small set of zip codes, such as the set listed in this table.

The "State" data class listed in the interface specification needs to be defined. Debbie can use a table to do that. She could include just the states or all USPS recognized abbreviations.

State	
Full Name	**Abbreviation**
North Carolina	NC
Massachusetts	MA
… and more	

The ZIP Code Lookup tests do not have to include any states that are not in this table.

Implementation Tests

Debbie can provide additional tests for an interface implementation. The initial test checks that the interface is working properly. She may also be concerned with the performance of the implementation. If it takes a long time to look up a ZIP Code, then the user experience will suffer. So she can specify the amount of time in a test. The time limits may differ based on the type of result (success or failure) and the particulars of the data being passed to the service.

Zip Code Lookup					
Street Address	**Line 2 Address**	**City**	**State**	**ZipCode?**	**Time?**
1 East Oak Street		Chicago	IL	60611	.01
101 Penny Lane		Danville	VT	05828	.01
1 No Place		Nowhere	OK	NotFound	.2

For a data-lookup style service, such as the ZIP Code, Debbie could also make up acceptance criteria for the completeness and accuracy of the data. The criteria can be specified in a table such as:

ZIP Code Lookup Qualities	
Completeness	**Accuracy Of What Is Available**
99.999%	99.9999%

Determining whether an implementation meets these criteria is a difficult task and beyond the scope of this book. However if you have two implementations of a service, you can at least cross-check them.

Comparing Implementations

In the case of ZIP Code Lookup, Debbie may be able to obtain two implementations of the interface. She may not know in advance what the correct results are. All she knows is that the results of the two implementations need to match. This is often the case when you have an existing system and you are re-writing the system to use another technology. The external behavior of the new system must match the existing system. So you first run one implementation and store the results. Then you run the second implementation and compare the results to those found in the first implementation.

Using tables to represent this comparison involves creating some type of variable. A variable is used to store the results of one action for use in a later action. You need to be able have a way to show when a value is stored in the variable and when it is retrieved. For the purposes of demonstration, we'll use a "→" symbol to show a value is stored into a variable and a "←" symbol to show the value is retrieved from the variable.[9] Debbie creates the following table for the first implementation of the interface:

Zip Code Lookup		**Implementation = Daves**		
Street Address	**City**	**State**	**ZipCode?**	
1 East Oak Street	Chicago	IL	→eastoakzip	
101 Penny Lane	Danville	VT	→pennylanezip	
1 No Place	Nowhere	OK	→noplacezip	

The ZIP Codes are stored in variables named "eastoakzip", "pennylanezip", and "noplacezip". Debbie then makes the next table for the second implementation.

Zip Code Lookup		**Implementation = USPS**		
Street Address	**City**	**State**	**ZipCode?**	
1 East Oak Street	Chicago	IL	←eastoakzip	
101 Penny Lane	Danville	VT	←pennylanezip	
1 No Place	Nowhere	OK	←noplacezip	

The ZIP codes returned by this implementation are compared to values stored in "eastoakzip", "pennylanezip", and "noplacezip". If a value does not match, an error is indicated. If an error appears, Debbie cannot be sure which implementation was wrong without further investigation.

If the number of comparisons was large, Debbie might not necessarily use a table to list the individual data items. She might create a module that did the comparison internally. So all she would list are the input values. If a mismatch occurred between the two implementations, the two answers could be shown in the ZIP Code column.

Zip Code Lookup Comparison		**Implementation = Daves** **Implementation = USPS**		
Street Address	**City**	**State**	**Daves ZIP Code?**	**USPS ZIP Code?**

[9] The methodology and symbols used for storing and retrieving values vary in each test framework.

1 East Oak Street	Chicago	IL		
101 Penny Lane	Danville	VT		
1 No Place	Nowhere	OK		

Note

Using two or more implementations and comparing them is a common design solution, particularly in critical systems. For example, in space flight, three computers calculate the required flight operations, such as when to turn on the rocket booster. The results of the three are compared. If they are all the same, the operation commences. If they all disagree, then "Houston, we have a problem." If two agree and one doesn't, then usually the majority wins and "Houston, we may have a problem."

A Brief Note on the User Interface

Debbie has not specified how the state is going to appear in the user interface. She knows that separating the business rules from the display makes for easier testing as shown in Chapter 14. There are at least four ways the state could appear on the user interface.

- A text box that accepted two character abbreviations for the state
- A text box that accepted the full name for the state.
- A drop-down list that contained the two character abbreviations for each state
- A drop-down list that contained the full name for each state.[10]

These are four display manifestations of the same requirement. They are not four different requirements. They differ in the user experience. In the drop-down version, the user cannot enter an incorrect state, but they may have to type more.[11] The selection of one is based on user quality feedback.

In any event, the field for the zip code should only allow either five or nine digits to be entered. These are the two valid lengths for US ZIP Codes. To allow other characters or lengths would cause unnecessary calls to the ZIP Code Lookup.

If Debbie had a field that allowed the user to enter a ZIP Code, she would check that ZIP Code against the one returned by ZIP Code Lookup. How she displays a mismatch is based on the desired user experience. The mismatch might show up in a dialog box, as a line at the top of the dialog, as a message next to the zip code, or as a different colored zip code.

One Bigger Level

The example is being pretty U.S centric. Debbie might need to do postal code matching for all the countries in the world. To keep things simple, she would have a master table of all the countries that breaks out to tables of tests for each individual country.

[10] As a resident of North Carolina, I highly prefer the two-character drop down list - Can you figure out why?

[11] Try to enter North Carolina in a drop-down that has full states. How many keystrokes does it take?

Country Breakout			
Country	**ISO Code**	**Input with**	**Validate with**
U.S.A.	US	US Address Form	ZIPCodeLookup
Canada	CA	CA Address Form	CAAddressValidator
	.. and many more		

A Reusable Business Rule

A business rule is something that is true regardless of the technology employed (paper, computer, etc.). The rule that a ZIP Code be valid for an address is true regardless of whether the envelope is printed on a laser printer or handwritten. Implementations of business rules should be exposed so that they can be used in multiple places, not just the mid-tier.

For example, the user interface may require business rule checking to allow errors to be identified in a more user friendly manner. The ZIP Code for customer address may be verified as part of the input process. To avoid duplication of functionality which would require duplication of testing, the user interface should use the same module as the mid-tier. The means for doing so are dependent on the technology involved, so are beyond the scope of this book[12]

Not only should reuse be considered with a single application. If a function such as ZIP Code Lookup is used by multiple applications, Debbie would put the component into an infrastructure or core system library. That would eliminate having to have tests applied to each application.

Summary

- Developer acceptance tests should be created for every service
- A common API should be created for services that multiple implementations
- Create performance tests for service implementations
- Create completeness and accuracy criteria for service implementation
- Create comparative tests for checking old versus new implementations
- Keep tests for services separate from tests for user interfaces
- Consider whether services are application services or core services

[12] You might use Ajax [Riordan01].

Chapter 18

A Simple System Grows

"The pure and simple truth is rarely pure and never simple."

Oscar Wilde

Debbie presents a more complex way of looking at the system. She introduces a model diagram and how it is represented in tables. She explains that the table representation is determined by Cathy.

A More Complex Example

Debbie starts off, "Cathy, we created tests for check-out and check-in that used a simple setup. The setup had a Customer table and a CD table. The CD table had an entry for each physical CD. Now there are probably some database oriented people out there who are trying to figure out the underlying database organization. If there were only one copy of a CD for every CD Album, then there is no redundancy in the data. And if we do not need to keep any rental history, but just the current state of the rental, we can do so as part of the CD."

"This simple organization worked when we were just getting started. But as we have discovered more about your domain and your needs, we can revise our understanding. Suppose that the CD Data for the production data looks like this."

CD Data					
ID	**Title**	**Rented**	**CD Category**	**Customer ID**	**Rental Due**
CD3	Janet Jackson Hits	Yes	Regular	007	1/3/2010
CD7	Janet Jackson Hits	No	Regular		

You'll notice that there is some duplication in this table. The title and the CD Category are the same for both CDs. That's because they are copies of the same album. We'd like to eliminate this duplication, but first we need to agree on some terms"

"People often talk about an artist releasing a new 'CD'. Note that's the same term as what we are currently calling a CD. We use the term CD for a physical copy of an artist's release. We can't use the same term for both the physical copy and the release or else we'll all get confused when we talk about a CD. So, Cathy, it's your call – what should we call these two concepts?" Debbie asked.

Cathy ponders, "Well, we've been using the term CD for so long; I think we should keep that as the name for the physical copy. I guess we could call the release either a Release or an Album. Sam and I always talk about getting a new Album. I don't think we ever call it a release. The artist releases an Album. So that's what the name should be."

"Alright," Debbie replies. "So let's make up a second table. We need to show a relationship between these two concepts – CD and Album. In programming land, we call that an association. Often a diagram is an easy way to visualize these associations. A common diagramming method is call Unified Modeling Language or UML for short. UML has so many features and options it takes a four-hundred page book just to show them all. I'm going to use just the simplest form since using too many symbols gets confusing [Wiki06], [Ambler01].

"We show the two entities – CD and Album in boxes in Figure 18.1. There are different ways to represent the association that a CD "is a copy of" an Album. We could just draw an arrow between the two boxes. Or we could put a representation of how many are on each end. The '*' says that there can be many copies of an Album and the '1' says that each CD is a copy of only one Album."

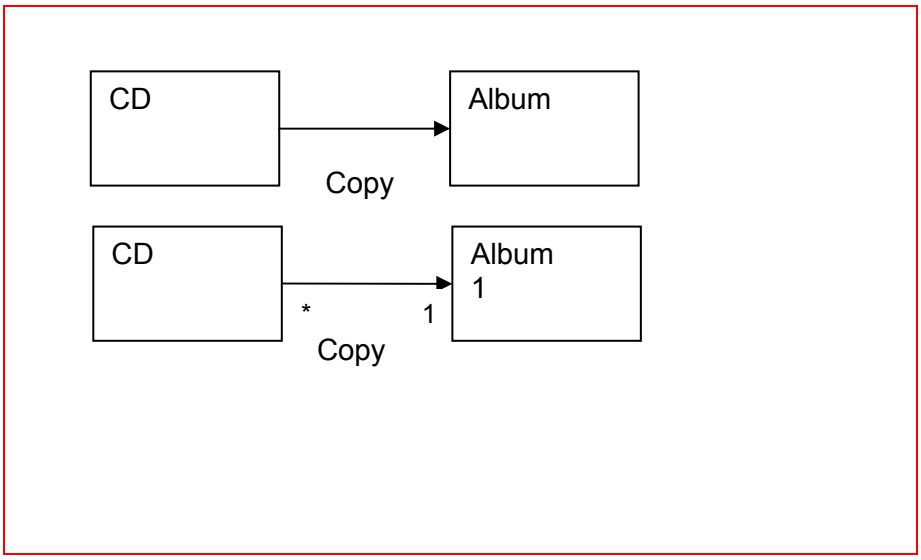


Figure 18.1 *Diagram of CD Album Relationship*

Debbie resumes, "Now how do we know that a CD is a copy of a particular Album? I sort of know the answer, but you are the subject matter expert and I want to use exactly the term you use without being influenced by what I say."

"Well," Cathy answers. "We order a copy by using the UPC code that's on the album cover. So that would be how the two are related."

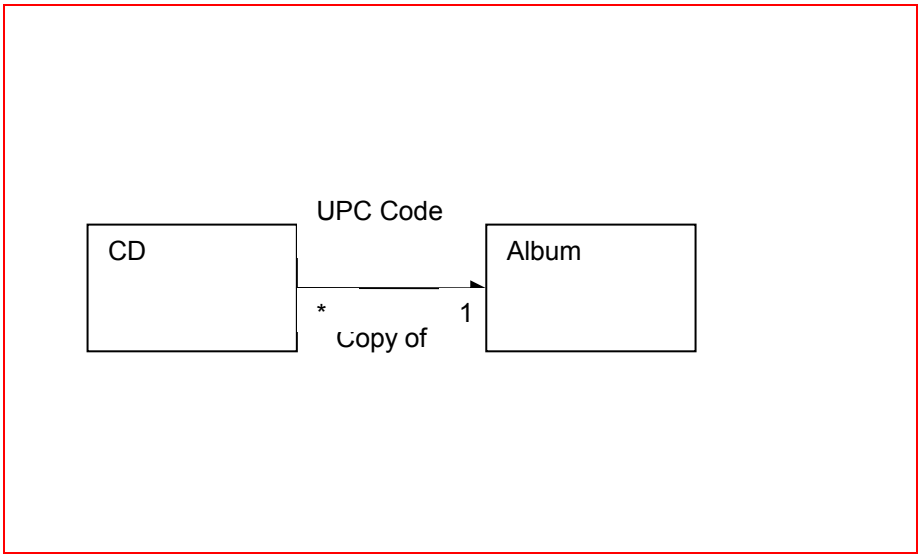


Figure 18.2 *Diagram of CD Album Relationship with Specific relationship*

Debbie continues, "We could show this relationship with two tables. In a little bit, I'll give you a another way."

Album Data		
UPC Code	**Title**	**CD Category**
UPC123456	Janet Jackson Hits	Regular

and

CD Data				
ID	**UPC Code**	**Rented**	**Customer ID**	**Rental Due**
CD3	UPC123456	Yes	007	1/3/2010
CD7	UPC123456	No		

Debbie resumes, "If we need to find the title or the category for a CD, then we look it up in the Album. Now if you need to keep track of rental history, we could create another relationship. We don't have a story yet that requires a history. But since we're talking about relationships, it seems appropriate to show one more example of how we can decrease the number of columns in our tables."

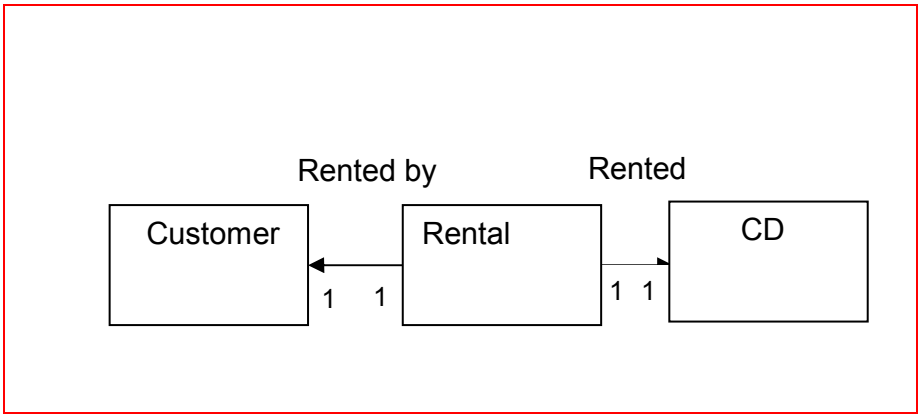


Figure 18.3 *Diagram of CD Customer Rental*

"What I have done is separate out the concept of being rented from the the CD. This separation makes a better design and can make for easier testing. With this new organization, the CD does not know it is rented or not. That information is now kept in the Rental Data. Note that if the CD is rented, then the Customer ID and Rental Due in Rental Data are valid. If the CD is not rented, then those two entries are meaningless. They might represent the last customer to rent the CD or they might simply be blank. Here's what the data looks like:"

CD Data	
ID	**UPC Code**
CD3	UPC123456
CD7	UPC123456

Rental Data		
CD ID	**Customer ID**	**Rental Due**
CD3	007	1/3/2010
CD7		

Customer Data	
Name	**ID**
James	007

Debbie asks, "Cathy, does this make sense so far? I'm going to do just a couple more manipulations. At any point, we can stop, since the purpose of the acceptance tests is for all of us to communicate together. Some of these tests might be used by Tom and me to see whether our database structure works. You would only see the tables we've discussed in the first section of this chapter.

Cathy replies, "So far, so good. How much farther are you going to go?"

"Just one more little step," Debbie answers. "And we'll hold off on the history part. Since the Customer ID and Rental Due are blank if the CD is not rented, we could make a convention that if the CD is not rented, an entry does not appear in the Rental Data. So if only CD3 was rented and not CD7, then we would have:"

Rental Data		

CD ID	Customer ID	Rental Due
CD3	007	1/3/2010

Debbie continues, "There's no sense including rows that do not have any useful data in them. So this table is even simpler"

Tom interjects, "We do not need to alter the previous tests in Chapter 10. We could present what's called a view that represents the data just as we described it before. The view decouples how we represent the data underneath from the way the business deals with the data. We leave it as an exercise to the technical reader of how the previous tables can be viewed as the original table:"

CD Data					
ID	**Title**	**Rented**	**CD Category**	**Customer ID**	**Rental Due**
CD3	Janet Jackson Hits	Yes	Regular	007	1/3/2010
CD7	Janet Jackson Hits	No	Regular		

Debbie resumes, "By breaking things down into smaller pieces, we cut down on the number of columns in each table. So there is less information to process. It can become easier to understand the tests, since the tables focus only on the information that is required for the test itself. But that's only if you, Cathy, understand the new tables. Otherwise, we stay with what we have. Tom and I might still have some tests using these shorter tables underneath to check out the database design. Here's how the test might look with the new tables."

Given a customer with a rental

Customer Data	
Name	**ID**
James	007

Rental Data	**Customer ID = 007**
CD ID	**Rental Due**
CD3	1/3/2010

When the clerk checks in the CD

Test Date
Date
1/4/2010

Checkin CD		
Enter	**CD ID**	CD3
Press	**Submit**	

Then CD recorded as not rented and the rental fee is computed. The fee

includes one late day.

Rental Data	Customer ID = 007
CD ID	Rental Due

Rental Fee				
Customer ID	Name	Title	Returned	Rental Fee?
007	James	Janet Jackson Hits	1/4/2010	$3

Rental History

Debbie starts off, "Now let's consider rental history. When a CD is returned, we could make an entry in a Rental History Data repository. Sam and you could use the data in this repository to do analysis of rentals. If we recorded history, the preceding test might have this as the output:"

Rental History Data			
CD ID	Customer ID	Rental Due	Returned
CD3	007	1/3/2010	1/4/2010

"Now this is the usually the easiest way to show history. If you recorded that the CD has been rented twice, the table would look like:"

Rental History Data			
CD ID	Customer ID	Rental Due	Returned
CD3	88	12/1/2009	12/3/2009
CD3	007	1/3/2010	1/4/2010

"If we are only going to connect the rental history to a CD, then you could show the history as a table embedded in another table[13]. This cuts down on the number of tables, but it does increase the size of a table. It's your call as to which is easier for you to understand."

CD Data		
ID	UPC Code	Rental History
CD3	UPC123456	Customer ID / Rental Due / Returned 88 / 12/1/2009 / 12/3/2009 007 / 1/3/2010 / 1/4/2010

[13] In Domain Driven Design Terminology, CD is the root of an aggregate.

Debbie continues, "You might also use an embedded table to show the individual parts of particular columns. For example, for a Customer, you could show the individual parts of a Name and an Address with contained tables:"

<table>
<tr><td colspan="3">Customer Data</td><td></td><td></td></tr>
<tr><td colspan="3">Name</td><td colspan="2">Address</td><td>Date Joined</td></tr>
<tr><td>First</td><td>Last</td><td>Prefix</td><td>Street</td><td>City</td><td>State</td><td>ZIP</td><td>03/04/2003</td></tr>
<tr><td>John</td><td>Doe</td><td>Mr.</td><td>1 Doe Lane</td><td>Somewhere</td><td>NC</td><td>99999</td><td></td></tr>
<tr><td colspan="3"></td><td colspan="2"></td><td></td></tr>
</table>

"Another way of representing contained data such as the address is to use a different organization of the table, such as:"

Customer Data		
Name		
	First	John
	Last	Doe
	Prefix	Mr.
Address		
	Street	1 Doe Lane
	City	Somewhere
	State	NC
	ZIP	99999
Date Joined		03/04/2003

Debbie continues, "The important thing is that you decide the most appropriate form for how the information is presented. Tom and I are responsible for converting the form into the appropriate code. However we might have a couple of suggestions for small changes that would make our job a lot easier."

Cathy smiles, "Considering that I'm paying you on an hourly basis, anything that makes your job easier makes it cheaper for me."

Summary

- Relationships between entities can be diagramed for ease of understanding.
- Entity relationships can be shown in multiple ways in tables.
- The preferred form of showing relationships is determined by the customer unit.

Chapter 19

A Still Larger System

"It always takes longer than you expect, even if you take Hofstadter's Law into account."
Douglas Hofstadter

We show how larger systems have more and different triads. Some projects do not require new customer acceptance tests. How to deal with a lack of acceptance tests is examined.

Larger Systems

Sam's system had just two people - Debbie and Tom - as developer and tester. Debbie was an omnipotent developer. She did everything from creating the overall architecture to designing the user interface to administering the database. Tom did all sorts of testing, from helping with acceptance test development to running performance testing tools to checking usability and performing exploratory testing. Teams for larger projects are sometimes composed of such ambidextrous individuals. But often the range of technology involved and the scope of the project do not allow for individuals to be able to cover the entire gamut of development and testing. Nor does a single customer as Cathy know every detail about what needs to go into the system. Subject matter experts specify the requirements in their own particular area of expertise.

The triad still exists, just with different people. It becomes the subject matter expert, the developer (or developers if you are doing pair programming), and the functional tester. They develop acceptance tests for the particular stories or requirements with which the expert is familiar.

For project with larger implementations, teams often have an architect or technical lead who is responsible for the overall structure of a large system. The system may encompass a single application or multiple applications.

For teams like this, you have another triad - the architect, developer, and tester (Figure 19.1). The architect, say Al, develops the organization of the system – its modules and their inter-relationships. Al assigns responsibilities to each module. The responsibilities are specified with acceptance tests, like the ones Debbie created in Chapter 16. The triad develops the tests collaboratively. Since all involved people are of a technical bent, the tests may incorporate many non-customer related terms. But everyone in the triad needs to understand the terms and their implications.

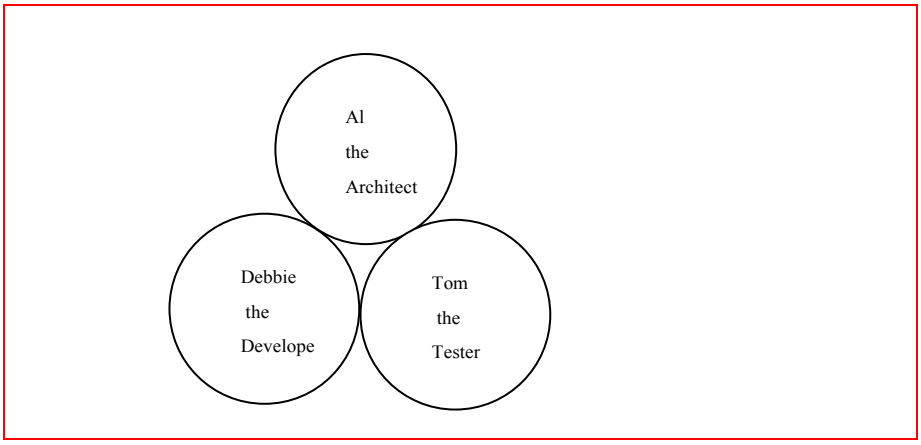


Figure 19.1 *The Technical Triad*

Some tests for the modules are derived from the customer acceptance tests, just as unit tests are derived from them. The process of deciding how many modules are required and which modules are responsible for fulfilling which parts of the acceptance tests is a major facet of the architectural design process. This process is covered in other books, e.g. [Fowler01]

One key that the team should focus on is to get an end-to-end system working soon after the start of the project. One customer acceptance test should be demonstrated on the system. The feedback from the ease or non-ease of developing for that test can yield helpful information to Al as to whether a particular architecture is suitable for the project. It also gives a baseline against which to measure additions. When a new story was implemented, a customer acceptance test may fail because an implementation changed its behavior. If the behavior represented by the test is still required, then there is an opportunity for Al to review and possibly revise the architecture. This is before much work has been spent writing code dependent on the architecture.

With larger systems, you may have a database architect, say Dana. Dana creates the persistent storage required by multiple applications. Dana ensures that there is no redundancy of information, such as storing a customer's address in two different places. A different triad – the developer, the tester, and Dana – work together to ensure that the developer has a way to persist all the information needed to solve the story that the developer is working on. The story could be one from the customer or one from the architect.

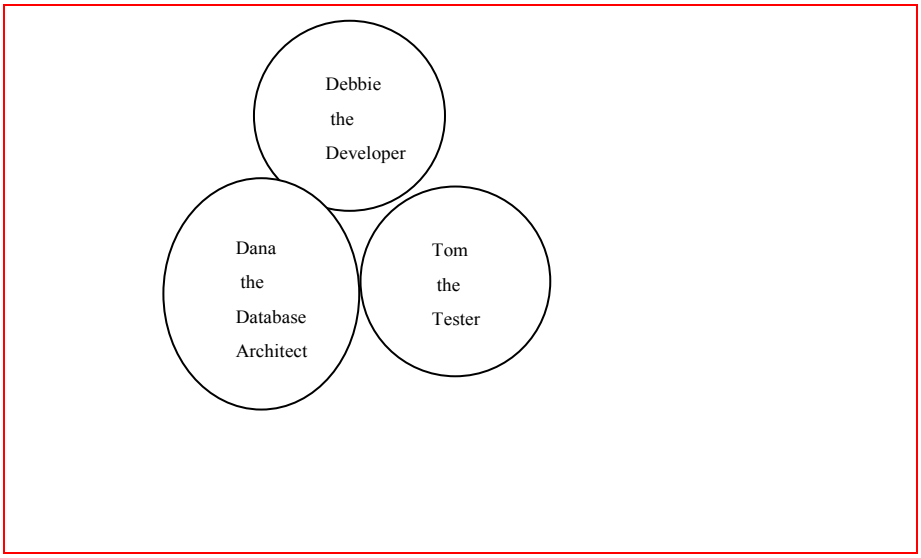


Figure 19.2 *Another Technical Triad*

As we have seen, the triad concept can work all the way up and down the chain, from an overall corporate software strategy down to the individual modules. The triad consists of the requestor, the implementer, and the tester who ensures that all bases are covered. The triad is meant as a minimum for the number of people in collaboration. You may have a quad, a penta, or larger group as required. Often the larger the group, the less the interaction and the less effective it can be. So limit the number to those actually required rather than those who just have a possible "want to know what's happening."

What's A Customer Story ?

Sam's business is booming. He now rents not just CDs, but electronic books, videos, games, and almost everything under the sun. He has bought up a number of competitors. So now Al has the work of keeping a large set of diverse programs working together. Al needs to combine systems so that the Counter Clerks will use the same system if they switch stores. The combination involves taking the data, such as customers, from one system, and converting it to another system. There are projects like these that may not necessarily involve new customer acceptance tests.

Data Conversion

Al and Dana, the database architect, work together on the data conversion project. This conversion is a mostly technical project, not a customer project. Al will be the one to write the acceptance tests for the conversion. He will specify the measures of the cleanliness of the conversion. For example, he may create the rules to determine that a customer who may reside on two systems is a duplicate. Al knows that Sam does not want a customer showing up twice on the converted system.

There may be some business issues, such as differences in customer status from one system to another. For example, Sam's business rule for allowing someone to reserve may be different than the business rule for an acquired competitor. You can keep track of where a customer came from and incorporate both. But that is Sam's call.

There should be no new customer acceptance tests, since they have already been created for the working system. The only exceptions would be tests that check that the business issues, such as customer status, have been appropriately handled.

Database Conversions

Dana, the database architect, has been informed that there is not going to be any more support for the database that Sam's systems use. Dana needs to convert to the next version, or perhaps to a different system. Is this a customer need? No – it's a technical issue. There are no new customer acceptance tests since the behavior of the system should not change. All acceptance tests can work as a regression suite. When the conversion is done, the tests should still run as before.

If a project involves lots of stories for which there are no customer acceptance tests, then it may well be a technical project. Often large technical issues such as database conversion are incorporated into customer-focused projects. The customer often does not have any knowledge of the underlying issues and therefore has no ability to provide acceptance tests. If this is the case, then the technical parts should be broken out into a technical project with a technical lead playing the part of the customer. A project such as an upgrade to new database or a new version of a language or a new operating system, are definitely technical infrastructure issues.

And If There Are No Tests?

You have a system which has been acquired for which there are no acceptance tests. No manual ones. No automated ones. And you need to make some changes. What can you do? Let's look at a couple of conditions. In one, you may acquire the system from a vendor and in the other, you may inherit it from an acquisition.

When you buy a vendor application, it may be configurable or customizable. Configurable means that you set up values to make the application run in your environment or with your set of data. The logic in the application use this configuration information to alter its operation in pre-determined ways. You may be able to add some additional features, such as a Word macro. But these macros make use of existing operations. Since all operations should have been tested by the vendor, then having acceptance tests is less critical.

Customizable means that you are provided with some source code for the system that you alter to make a system work for your particular purpose. The code provides some existing behavior that you are changing. In this case, you should ask the vendor for their acceptance tests – either manual or automatic. Their response might be: [14]

- Don't have any
- Have some, but all manual
- Have full set, but all manual
- Have full set, some manual, some automated
- Have full set completely automated

[14] There are some gray areas which might be considered either configurable or customizable, but that's a discussion for beers after work.

If they don't have a full set, ask them how they know that the system works. It could be they have a full set, but contractually they are not required to provide them to you. If you can't find another vendor that will provide acceptance tests, then you are in the same situation as from an acquisition.

You have to change an acquired system. The system has few or no acceptance tests in the area that you wish to change. Chances are that the system has not be designed to be tested. [Feathers01]

Create acceptance tests for the functionality you are going to change. Inject the tests underneath the user interface if you can. Otherwise, run the tests through the user interface. Automate the tests if possible. Every test should pass as it documents the current working of the system. These tests will now run as a regression test. Make up an acceptance test for the change you are going to make. Determine which of the current tests, if any, should fail once the change is made. Then implement the change and test.

Legacy Systems

One common issue with legacy systems is the lack of tests – both external acceptance tests and internal unit tests. And often when there are external acceptance tests, they are not automated. Making changes in the system and ensuring that the changes do not have unintended effects is difficult.

Before making a change, you can create acceptance tests around the portion of the system which is involved with the change. The acceptance tests document how the system currently works. Then write acceptance tests for how the system should work once the change is implemented. Initially they should fail or otherwise, the system is already doing what the change request asks for. If the change should break an existing test, note that as well.

In many instances, you may have to write the tests as user interface tests. If it's possible, automate these tests. If the system design allows it, write the tests to the mid-tier layer and automate those tests. Then proceed with the change. When your new tests pass and any identified as "should break" fail, then the system has been changed correctly.

Suppose you don't have tests around every functional piece of the system. With the tests around the part of the system you are changing, you ensure there are no side effects in that part. But you cannot be sure that the change has not affected anything else.

An acquaintance of mine asked me to give an estimate on converting a web application from a commercial application server to an open-source one in order to save licensing fees. The application had been coded by a third party. I examined the code and determined that there were only a very few vendor-specific methods that would need to be changed. I gave him the estimate. He said it seemed reasonable.

I said that it did not include fixing any bugs that currently existed in the system. He agreed to that. I suggested that my responsibility would be over when the converted system passed all the automated acceptance tests. He hesitated and replied that there were no automated acceptance tests. I said then my responsibility would end when the system passed all the manual acceptance tests. He again paused and acknowledged that there were no acceptance tests at all for the system.

I looked at him and stated that I could then have a converted system that passed all the acceptance tests that afternoon for half the price. He smiled. I then proposed that I would help develop acceptance tests for the current system, since his staff did not have time to do so. That was agreed to, but the total cost of creating the tests and doing the conversion far exceeded the license fees, so the project never got off the ground.

Summary

- Larger teams have more triads
- Focus on getting an end-to-end system passing the simplest tests
- Some projects are developer related and should have developer-created acceptance tests
- If there are no acceptance tests for the portion of a system that needs to change, create acceptance tests before making the change

Chapter 20

Test Setup

" It's not what I do, but the way I do it. It's not what I say, but the way I say it."

Mae West

We discuss the tradeoffs between using an individual setup for tests and using a common setup. Concerns about test order and persistent storage are explored.

A Common Setup

You may have noticed that the setup for several of the triad's tests started to be repetitious. The tests all need customers and CDs to rent or return. There is a tempting reason to place the repetitive setup into a common setup. There is a classic tradeoff between decreasing duplication by the common setup and increasing the possibility that altering the common setup causes failing tests.[15] A common setup for all tests might be:

Customer Data		
Name	**ID**	**Credit Card Number**
James	007	4005550000000019
Maxwell	88	4005550000000028

Album Data		
UPC Code	**Title**	**CD Category**
UPC123456	Janet Jackson Hits	Regular
UPC000001	Beatles Greatest Hits	Golden Oldie

CD Data	
ID	**UPC Code**
CD2	UPC000001
CD3	UPC123456
CD7	UPC123456

This data forms a "test bed" for the other tests. If you make any changes to this setup, such as changing James's ID or Credit Card Number, then tests that depend on those values will fail. They will

[15] This is a example in the test world of the "Splitters versus Lumpers" prefactoring guideline for design. [Pugh02]

fail not because the behavior of the system changed, but because the test may now be specifying an invalid behavior. For example, the Card Charge would have James's original credit card number on it, but James's number has changed.

The Rental table could be a shared setup. However it has more potential states than the CD Disc and the CD Title tables. So you might have a shared table that looks like this:

Current Rentals		
CD ID	**Customer ID**	**Due Date**
CD12	007	1/1/2010
CD6	88	1/2/2010

Alternatively, the data in this table could be expanded, so that the same setup could be used for the test of the "Customer Limit On Simultaneous Rentals" business rule.

Current Rentals		
CD ID	**Customer ID**	**Due Date**
CD12	007	1/1/2010
CD6	88	1/2/2010
CD20	88	1/3/2010
CD 21	88	1/4/2010

Customer ID 007 could be used for a regular check-in test. Customer ID 88 could not be used for a regular test, since Maxwell already has three CDs rented. Keeping track of which customers are suitable for which tests requires some discipline.

This common setup could be run for each test to get a clean start – "a Fresh Fixture" as Gerard Meszaros terms it [Meszaros01] -- or it could be run at the end of a series of tests that do not affect the setup. In this particular case, if the tests that are run add more rentals to ID 007, then James may reach the rental limit, and subsequent tests that assume he is not at the limit will fail.

Some Amelioration

There are some practices you can do to ameliorate potential problems. First, never change existing data in the setup. Add to the setup if a different entity is needed. For example, if you need a customer with different characteristics, add another customer. Some teams give names to the entities such as customers that represent the kind of entity they are dealing with, such as "Big Spender", "Prompt Returner", and so forth. For example, instead of ID 88 being named Maxwell, he might be named "Customer Who Reached Limit" if the previous Rental table was used.

Another way of handling the issue is to not have a common setup that is used for every test, but ones that are common to a group of tests. The number of tests that a change in the setup can potentially affect is limited.

Still another way is to use variables in the setup as shown in Chapter 17. For example, you might define a variable to contain James's credit card such as:

4005550000000019 → JAMES_CREDIT_CARD

In the tests, you would make a reference to JAMES_CREDIT_CARD rather than the number itself. [16] For example in the setup, you might use:

Customer Data		
Name	**ID**	**Credit Card Number**
James	007	←JAMES_CREDIT_CARD

Still another way of making tests less dependent on the setup is to use relative results. For example, you might be testing the ability to add a customer, say Napolean Solo. Using this setup and an absolute result, the test would check the result to see that there are two customers in the Customer Data. Using a relative result, the test would first determine the current number of customers and then check to see that the number of customers after the addition of Napolean Solo was the previous number plus one.[17] In both cases, the test would check that Napolean Solo was a customer.

Test Order

In most acceptance test frameworks, you have control over the sequence in which tests are run. If a test has no side effects, then the order in which it is run relative to other tests is unimportant. A side effect is a change to the state of the system or an output to an external repository, as shown in Chapter 8. If a test does have a side effect, such as deleting a customer, you may need to run the tests in an order such that the deletion comes last.

You may be able to take advantage of complementary side effects. Suppose one test adds a customer and another test deletes a customer. If you run these tests one right after the other, the side effects should cancel out.[18]

Running tests in a particular order, just like depending on a common setup, is something that should be well justified and documented. Otherwise maintenance of the tests may accidentally change the order and cause tests to break.

Persistent Storage Issues

Often tests include altering entries in persistence storage such as a database. Even if the tests do not use the same entities such as customers, they may leave the database in a state in which it is not ready for the tests to be run again. You need to restore the database to its original state.

Depending on your test environment, you may have multiple technical solutions to this problem. You can simply restore the database from a backup copy. Although this takes some time, it may be small relative to amount of time for all the tests. Alternatively, you could execute the tests within a virtual machine. The virtual machine is closed upon test completion and a new clone of the virtual machine is used for the next test.

A database on a test platform may consist only of records that the test uses. A database on a pre-production platform may have million of records. It would take considerable time to restore the entire database. You could create a procedure that backups and then restores just the entities that have been

[16] This is an example of the DRY Principle – "Don't Repeat Yourself" from the Pragmatic Programmers [Hunt01] . This also known asr the Once and Only Once Principle. A corollary of this principle is Shalloway's Law. If there are N places where a change has to be made, Shalloway will find N-1 of them.
[17] A relative result is also referred to as a delta. So a test that uses a relative result is called a delta test.
[18] A log of operations can include entries on both of these operations. So the side effect of logging is not cancelled out.

affected by the test. Alternatively you could have a process that eliminates all traces of an entity such as a customer. Then the tests could create that entity and use it for further tests.[19] Another approach to restoring persistent data to its original state, particularly for tests that add to the data, rather than modify it, is to timestamp every record. When the test is over, all records whose timestamp is equal to or after the time at the beginning of the test are deleted. You could also create a log of records that have been modified during the tests and restore the original record once the tests have completed.

If you don't have control of the database, what you can do is document what the setup should be, so that a test will fail not because of the action, but because of the state that the system is in. If the 'Given' part is not given, then the test should fail. For example, suppose you are testing adding a customer and the customer you want to add is ID 99. You need to ensure that the the customer does not already exist. So you place this in the "Given" section. If you have a lot of these conditions, you might separate them from the "Givens" into an "Assume" section.

Customer Data		
Name	**ID**	**Exists?**
Agent	99	No

Summary

- Use a common setup to create a "test bed" for further tests
- There is a tradeoff in eliminating redundancy and causing dependencies between a common test setup and individual test setups
- Be cautious of tests that have side effects that alter conditions for other tests
- If using shared resources, document assumptions as to the condition of those resources.

[19] This is like what happened to George Bailey in *That's a Wonderful Life*.

Chapter 21

Test Presentation

"Ring the bells that still can ring
Forget your perfect offering.
There is a crack in everything,
That's how the light gets in."

Leonard Cohen

There is no perfect way of writing a test or a table. Alternative ways are presented here.

Customer Understood Tables

The key in selecting the form of a table is to pick the one that the customer unit most easily understands. For example, here is the business rule table for rental charges.

CD Rental Charges			
Category	**Standard Rental Days**	**Standard Rental Charge**	**Extra Day Rental Charge**
Regular	2	$2	$1
Golden Oldie	3	$1	$.50
Hot Stuff	1	$4	$2

There are various ways the tests for this table could be documented. You could have a standard calculation style table, such as:

Rental Charges		
Type	**Days**	**Cost?**
Regular	3	$3
Golden Oldie	3	$1
Hot Stuff	3	$8

Alternatively, you could have an individual table for each computation, as:

Rental Charges	
Type	Regular
Days	3

Cost?	$3

Rental Charges	
Type	Golden Oldie
Days	3
Cost?	$1

Rental Charges	
Type	Hot Stuff
Days	3
Cost?	$8

There are forms of tables with labels that make the test read almost like a sentence. [20]

Rental Charges						
Rent CD	With Category	Regular	For Days	3	Charge should be	$3
Rent CD	With Category	Golden Oldie	For Days	3	Charge should be	$1
Rent CD	With Category	Hot Stuff	For Days	3	Charge should be	$8

You do not need to use the same style table in all tests. The triad should select the one most appropriate to the behavior being tested. Any disagreement should be settled by the customer unit.

Table Versus Text

The tests in this book have used tables to indicate the setups, actions, and expected results. If the customer prefers, the tests could be expressed in pure text, such as:

Given a CD that is a Regular, when it is rented for 3 days, the charge should be $3.
Given a CD that is a Golden Oldie, when it is rented for 3 days, the charge should be $1.
Given a CD that is a Hot Stuff, when it is rented for 3 days, the charge should be $8.

Specifying Multiple Actions

Sam's application has the business rule that a customer should not be allowed to rent another CD after that customer exceeds the rental limit. You could show this test as a sequence of action tables.

[20] This is a DoFixture table from the Fit Library by Rick Mugridge [Cunningham01]..

Given these rentals:
First rental

Start	**Checkout**	
Enter	**CD ID**	CD1
Enter	**Customer ID**	007
Press	**Checkout**	OK

Second rental

Start	**Checkout**	
Enter	**CD ID**	CD2
Enter	**Customer ID**	007
Press	**Checkout**	OK

Third rental

Start	**Checkout**	
Enter	**CD ID**	CD3
Enter	**Customer ID**	007
Press	**Checkout**	OK

Then the next rental should fail
Fourth rental

Start	**Checkout**	
Enter	**CD ID**	CD4
Enter	**Customer ID**	007
Press	**Checkout – Status**	Rental Limit Exceed

Alternatively, you could specify all the actions in a calculation style table. This reduces the size of the test and can make it more understandable to all members of the triad.

Multiple Checkouts			
CD ID	**Customer ID**	**Checkout Status?**	**Notes**
CD1	007	OK	1st rental
CD2	007	OK	2nd rental
CD3	007	OK	3rd rental
CD4	007	Rental Limit Exceed	4th rental

In the first version, the notes appear before each individual table. In this version, those notes are included in the table to explain each step in the test.

Still Another Table

The tables in the test can represent whatever is most appropriate for the application. If you were creating an acceptance test for an application that solves Sudoku puzzles, you might have a input table that looked like:

1			4			7		
	2			5			8	
		3			6			9
4			7			1		
	5			8			2	
		6			9			3
7			1			4		
	8			2			5	
		9			3			6

The output table would look like:

1	6	5	4	9	8	7	3	2
9	2	4	3	5	7	6	8	1
8	7	3	2	1	6	5	4	9
4	9	8	7	3	2	1	6	5
3	5	7	6	8	1	9	2	4
2	1	6	5	4	9	8	7	3
7	3	2	1	6	5	4	9	8
6	8	1	9	2	4	3	5	7
5	4	9	8	7	3	2	1	6

You might also have a test where the puzzle has no solution or where the puzzle has multiple possible solutions to see if the results match your expectation. A tester like Tom might have a puzzle that has thousands of possible results. He might create one that has only a single digit in it and see how long it takes to get all the possible solutions.

The key here is that the form of the test should be as compatible as possible with the way the customer unit deals with the functionality.

Summary

- Use the form of the table that is easiest for the customer unit to understand.
- If a standard table form is unsuitable, create a table form that is more appropriate for the test.

Chapter 22

Test Evaluation

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

Edsger Dijkstra

This chapter describes the characteristics of good tests. Characteristics include customer understandable, small, non-redundant, and not fragile.

Test Facets

Here is a list of things to look for in tests. Some of these came from Gerard Meszaros [Meszaros01], Ward Cunningham, and Rick Mugridge [Cunningham01]. Overall, remember that the tests represent a shared understanding between the customer, developer, and testing units.

Customer understandable

The test should be written in the customer terms (ubiquitous language) (Chapter 24). The tables should represent the application domain. If the standard tables shown in the examples are not sufficient, make up tables that are understandable by the user (Chapter 21). Use whatever way of expressing the test that most closely matches the customer's way of looking at things. Try multiple ways to see which way is most suitable to the customer.

Use what is easiest for the customer. If a basic action table is not as understandable, add graphics to make it look as a dialog. If what looks like a printed form is needed for understanding, rather than just the data, use that as expected output. You can also have a simple test for the data for easier debugging of what might be wrong with the printed form (Chapter 12)

Unless the customer wants values, use names. For example, use Good and Excellent as customer types, not customer types 1 and 2.

Spell Check

Tests should be spell-checked. Tests are meant for communication. Misspelled words hinder communication. The spell-check dictionary can contain triad agreed upon acronyms and abbreviations. These acronyms and abbreviations should be defined in a glossary.

Idempotent Tests

Tests should be idempotent. They should work the same way all the time. Either they consistently pass or consistently fail. An erratic test does not work the same way all the time. Types of erratic tests are Interacting Tests that share data and Unrepeatable Tests for which first and following executions are different because something in the state has changed. Paying attention to setup (Chapter 20) can usually resolve erratic tests.

Fragile Tests

Fragile tests are sensitive to changes in the state of the system and with the interfaces it interacts with. The state of the system includes all code in the system and any internal data repositories.

Handle sensitivity to external interfaces by using test doubles as necessary. (Chapter 11). In particular, the clock should be controlled through a test double. Random events should be simulated so that they occur in the same sequence for testing.

Changes to a common setup can cause a test to fail. If there is something particular that is required for the test, then the test could check for the assumptions it makes about the condition of the system. If the condition is not satisfied, the test fails in setup, rather than in the confirmation of expected outcome. This makes it easier to diagnose why the failure occurred.

Tests should only check for the minimum amount of expected results. This makes them less sensitive to side effects from other tests.

Confirm the Environment

Many programs require that the platform they are installed upon meet a particular requirement, such as a particular version of an operating system. When the program is installed, the installation process verifies that those requirements are met. If not, the installation terminates.

This approach is more user friendly than letting a program be installed and then the program failing because the environment is wrong. However the program makes an assumption that the environment does not change after it is installed. Sometimes the installation of another program changes the environment and causes the first program to fail.

To be less fragile, the program should confirm the required environment every time it starts up. If it is not as expected, the program should notify the user with an error message. That makes it much easier for a user to determine what the problem is than a "This program had a problem" message.

Test Sequence

Ideally tests should be independent so they can run in any sequence without dependencies on other tests. However, as previously noted, you often have required workflows. You then must run acceptance tests in the workflow sequence.

To ensure that tests are as independent as possible, ensure that a common setup (Chapter 20) occurs when necessary. As noted in that chapter, the setup part of a test ("Given") should ensure that the state of the system is examined to see that it matches what is required by the test. Internally this can be done by either checking that the condition is as described or by making the condition to be that which is described.

Workflow Tests

Workflow tests are sequences of tests that need to be run in a particular order to demonstrate that the system processes the sequence correctly. Try to have a happy path and an exception path in the workflow. For example with the CD rental limit of 3, have a test for renting 3 CDs (the happy path) and 4 CDs (the exception path).

Not all exceptions have to be tested if they result in the same workflow, just different output values. Variations of business rules or calculations, unless they affect the workflow, should be tested separately.

Have at least one positive and one negative test. For example, have a test with a Customer ID that is valid and one that is not valid.

Use case tests should have a single action tables in them. Workflows which have multiple use cases within them may have multiple action tables in their tests. Try not to have too many complicated workflow tests.

Test Conditions

A set of tests should abide by these three conditions[21]:

1. A test should fail for a well-defined reason (that is what the test is checking). The reason should be specific and easy to diagnose. Each test case has a scope and a purpose. The failure reason relates to the purpose.
2. No other test should fail for the same reason. Otherwise you may have a redundant test. You may have a test fail at each level – an internal business rule test and a workflow test that uses one case of the business rule. If the business rule changes and the result for that business rule changes, then you will have two failing tests. Tests should not be redundant. You want to minimize overlapping tests. But if the customer wants to have more tests because they are familiar or meaningful to him, do it. Remember the customer is in charge of the tests.
3. A test should not fail for any other reason. This is the ideal, but often hard to achieve. Each test has a three part sequence – Setup, Action/Trigger, Expected Result. The purpose of the test is to ensure that the actual result is equal to the expected results. A test may fail because the setup did not work or the action/trigger did not function or that the actual results were not returned correctly.

Separation of Concerns

The more that concerns can be separated, the easier it can be to maintain the tests. With separation, changes in the behavior of one concept do not affect other concepts. Here are some concepts that can be separated, as shown earlier in this book

- Separate business rules from how the results of business rules are displayed. (Chapter 14)
- Separate the calculation of business rule, such as a rating, from the use of that business rule. (Chapter 13)
- Separate each use case or step in a workflow. (Chapter 8)
- Separate out validation of an entity from use of that entity. (Chapter 16).

[21] These came from Amir Kolsky.

You can have separate tests for the simplest things. For example, the Customer ID formatting functionality needs to be tested. The test can show the kinds of issues that the formatting deals with. If the same module is used anywhere a Customer ID is used, then other tests do not have to perform checks for invalid Customer IDs. And if the same module is not used, then you have a design issue.

ID Format		
ID	**Valid?**	**Notes**
007	Y	
1	N	Too short
0071	N	Too long

Test Failure

As noted before in Chapter 3, a failing test is a requirement. A passing test is a specification of how the system works. Initially, before any implementation is created, every acceptance test should fail. If not, then you need to examine any acceptance test that passes. You should determine why it passed.

- Is the desired behavior that the test checks already covered by another test? If so, then the new test is redundant.
- Does the implementation already cover the new requirement?
- Is the test itself not testing anything? For example, the expected result may be the default output of the implementation.

Test Redundancy

You want to avoid test redundancy. Often that may occur when you have data dependent calculations. For example, here are the rental fees for different category CDs that was shown in Chapter 10.

CD Rental Charges			
Category	**Standard Rental Days**	**Standard Rental Charge**	**Extra Day Rental Charge**
Regular	2	$2	$1
Golden Oldie	3	$1	$.50
Hot Stuff	1	$4	$2

Here were the tests that were created.

Rental Charges		
Type	**Days**	**Cost?**
Regular	3	$3
Golden Oldie	3	$1
Hot Stuff	3	$8

Do you need all these tests? Are they equivalent? They all use the same underlying formula (Cost = Standard Rental Charge + (Number Rental Days – Standard Rental Days) * Extra Day Rental Charge).[22] When the first test passes, the other tests also pass.

What if there are lots of categories? Say there are 100 different ones that all use this same formula. That would be a lot of tests. But if Cathy the customer wants to see them, that's her call.

Acceptance Tests One Part of the Matrix

Acceptance tests are not the whole picture. They go hand in hand with requirements. There are the usability, performance, exploratory, and unit tests as shown in the matrix in Chapter 3.

Outside-in development provides a context. The set of acceptance tests for a story gives a context for the unit tests for the implementation. You do not need to translate all acceptance tests into unit tests. That would be duplication.

On the other hand, the acceptance tests can be implemented using unit test frameworks, such as xUnit. It depends on the collaboration between you, the customer, and the tester. The xUnit tests should be readable as shown in the example in Chapter 4, so that they can be matched with the customer's expectations.

If you don't have a customer providing you examples and you have a technical tester, then you might as well code all acceptance tests in xUnit. But if neither of those conditions apply, then having readable tests provides a double check.

Other Points

The following points may not fit neatly into the previous sections, but they are important to consider as well.

- Organize tests with the setup (Given) / action (When) / outcome (Then) sequence
- Only have the essential detail in a test to keep it simple.
- Avoid lots of input and output columns. Break it into smaller tables or show common values in the headers. (Chapter 18)
- Minimal maintainable – unless the desired behavior has changed, the test should not have to change, regardless of any implementation changes underneath.
- As much as practical, 100% of the code should be covered by tests – either acceptance tests or unit tests. There may be some exceptional conditions that are difficult to reproduce in a end-to-end pre-production platform.
- Test logic should not be in the production code. Tests should be completely separable from the production code.
- Automate the tests so that they can be part of a continuous build. See Appendix 2 for examples of automation.
- Tests and automation should be developed separately. Understand the test first and then explore how to automate it

[22] Depending on how the code is written, a single test might provide 100% code coverage. If you have that coverage, do you need more tests?

Summary

- Acceptance tests should be customer readable
- Avoid fragile or erratic tests by paying attention to setup and test doubles
- Avoid test redundancy

Chapter 23

Using Tests for Other Things

"By fighting, you never get enough, but by yielding, you get more than you expected."

Lawrence G. Lovasik.

Acceptance tests define the functionality of a program. But they can be use for more than just that – measuring doneness, estimating, and breaking down a story.

Uses of Acceptance Tests

Acceptance tests are a communication mechanism between the members of the triad. They clarify the customer requirements. They are a specification of how the system works. But they also can be employed for other purposes. They are a measure of how complete an implementation is; a means of estimating the effort to implement a story, and a method for story breakdown into smaller stories.

Degree of Doneness

If there are multiple acceptance tests for a business rule or a story, the ratio of successful tests to the total number of tests can provide a rough guide to how much of the story has been implemented. For example, if you have ten acceptance tests and three are passing, the story is "about" thirty-percent complete.

It is possible that the "worst" test case was saved for last. So the effort to implement that story represents more than its fair share of the total effort. That is why this is only a guide to doneness, rather than being an exact measure.

Estimation Aid

An estimate may be required for implementing a story in many environments. The number and complexity of the acceptance tests can be a rough guide to the effort required to implement a story. This requires previous completion of stories against tests. The tests can be compared to the tests run against the completed stories. You can develop your own heuristics as to how the number and complexity influence the effort. Large custom setups ("Givens") and numerous state changes ("Thens") usually imply a much larger effort than tests with small setups and few state changes.

Breaking Down A Story

You may need to break down a story into smaller stories for the purposes of fitting stories into iterations. Mike Cohn [Cohn01] gives some ways to break a story down. For example, you can start with a basic user interface and then add more bells and whistles such as images. As Debbie did, you can use keystrokes to enter the Customer ID and then add bar code reading later. You could implement something manually, such as calling users about overdue rentals, and later automate it with robot calling.

You could have one story to gather information, such as finding out how to do credit card processing, followed by another story to implement that processing. You can do something simple, such as a single check-out per CD and then have a story for checking out multiple CDs at once.

In addition, you can use acceptance tests as a breakdown mechanism. The test for each scenario of a use case can become a separate story. A complex business rule can become a separate story.

For a particular calculation, you can limit the number of different inputs that need to processed by eliminating some inputs and assuming default values for them. You could limit the number of different calculations by limiting the number of rows in a table.

With the address example in Chapter 17, the tests themselves suggest a way to breakdown the story. The tests for a U.S. address are separate from the tests for a Canada. Therefore U.S. address verification can be a different story than Canadian address verification.

Developer Stories

Another reason for breaking a story down is so that multiple teams can help implement it. Chapter 16 on developer acceptance tests covered how Debbie created acceptance tests for user interface components and functional modules. If a story has to be broken down, then acceptance tests should be created for each of the sub-stories. The acceptance tests help decouple the stories by clarifying the responsibilities of each of the stories. They provide doneness criteria for each story. It is far more effective for distributed teams to work on decoupled stories than to work on tasks for the same story. [Eckstein01]

Tests As a Bug Report

For non-trivial bugs, a passing test can be written as documentation of the bug. The definition of a non-trivial bug is something more than a misspelled word, a bad color, or an unaligned dialog box. The discoverer of the bug should create or help create two tests – an "un-acceptance" test that passes with the bug and an acceptance test that fails. The un-acceptance test demonstrates the behavior that is unacceptable, that is, the bug. The acceptance test that fails shows that the desired behavior has not been achieved.

Suppose you were coding the discount example in Chapter 4 and you did not have an acceptance test. If a bug was reported, you would create both an un-acceptance test and an acceptance test. To ensure that the bug fix did not affect other behavior, you may include test cases that are currently passing. Here are examples:

Discount Un-acceptance Test			
Item Total	**Customer Rating**	**Discount percentage?**	**Notes**
10.00	Good	0	
10.01	Good	1	
50.01	Good	5	Bug
.01	Excellent	1	
50.00	Excellent	1	
50.01	Excellent	5	

Discount Acceptance Test			
Item Total	**Customer Rating**	**Discount percentage?**	**Notes**

10.00	Good	0	
10.01	Good	1	
50.01	Good	1	correct
.01	Excellent	1	
50.00	Excellent	1	
50.01	Excellent	5	

When the bug has been fixed, you can eliminate the un-acceptance test since it is a specification of how the system should not work.

Root Cause Analysis

If a bug like the preceding appears, you have an opportunity to do root-cause analysis. There are web pages that give detailed explanations of how to do root-cause analysis. [Systems01] [Wiki07]

What you are looking for is the reason why the bug appeared. Is it something in the process itself or was it a random event. Was the case in production not covered by a test case and if so, why not? Were the data values not expected? For example, the program assumed a value between 1 and 100, but the value in production was 101. Since an acceptance test represents a requirement, then the need for this acceptance test after the code has been implemented may represent a missed requirement. An acceptance test that has the wrong values is a misinterpreted requirement.

The missed or missing requirement may be traced to a random event, such as "we had to get this done in four hours before release." Alternatively, it may be traced to a common cause, such as "the customer never collaborates with us before we start implementing." [23]

For common causes, determine how you can eliminate them, see Chapter 26.

Production Bugs

One of the most important measures for a team process is the number of bugs that escaped to production. You should examine the root cause or causes of each escaped bug. Examining why bugs escaped may lead you to discover how to prevent more of them from escaping in the future.

Summary

- You can use acceptance tests as:
  - A rough guide to story doneness
  - A rough way to estimate relative story effort
  - A way to break up stories
- Distributed teams that break up a story should have acceptance tests for their part of the story

[23] If the "we had to get this done in four hours before release" occurs more than once, then it is a common cause, rather than a random event

- Examine the root cause of why an acceptance test was missed or incorrect and if possible change the process to eliminate the cause.

24

Context and Domain Language

"England and America are two countries separated by a common language"

George Bernard Shaw (or Bertrand Russell or Oscar Wilde)

Communication requires a common language. During the collaboration in the creation of acceptance tests, a common language emerges.

Ubiquitous Language

Domain Driven Design [Evans01] refers to the ubiquitous language. The language is the manner in which the customer and developers talk about a system. The language arises from explanations given by a customer or subject matter expert explanations about the entities and processes in a system. The language transforms itself and becomes more refined as developers and customers discover ambiguities and unclearness.

The language evolves during the collaboration on the requirements and the tests, as shown in Chapter 6. Contributions to the language come from the column names in tables, the names of use cases and their exceptions, and the business rules. Each term in the language should be documented with a one sentence description that is provided by the customer unit and understood by the developer and tester units. The customer unit leads the terminology effort, but the developer unit can suggest that terms are unclear, ambiguous or redundant.

For example, the triad referred to entities as Customer, CD, Rental, and Album. They could be defined with single sentences as:

- A Customer is someone to whom we rent CDs..
- An Album is an artist's release.
- A CD is a physical copy of a Album that gets rented.
- A Rental contains the information on a CD that is rented.

You could use tables to define the terms. For example, the discount example in Chapter 4 referred to the terms 'Customer Type' and 'Item Total'. The Customer Type can be defined by a table, such as:

Customer Type	
Name	**Meaning**
Regular	A common customer
Good	One we want to keep

Excellent	One we will cater to their every whim

Order Total might be part of a larger picture table. You might show relationships to other entities if that clarifies the picture.

Order Fields		
Name	**Meaning**	**Formula**
Item Price	What we charge for the item	
Order Item	Item on an order	
Item Quantity	Count of how many of an item is ordered	
Item Total	Total price for a single item	Item Price * Item Quantity
Order Total	Total price for all items on an order	Sum of all Item Totals

The triad should agree not only on the meanings of entities, but also their identity and continuity. Identity is whether two entities are the same. For example, if a Rental Contract is reprinted, it represents the same entity as the first printout. However a Credit Charge that is resubmitted may be construed to be a new charge and the customer will get double-billed. If a customer returned two CDs within a very short time period, there could be two legitimate charges that looked exactly the same. The credit processor might interpret that as a duplicate billing and reject the second.

Continuity is how long should an entity persist. If customers want their rental history completely private, then a Rental entity should only persist until it is complete. That is, until the customer has checked-in the CD. On the other hand, if Sam wants the rental history to determine favorite CDs for a customer so he can offer them new releases in that same genre, then the Rental should be persistent.

Two Domains

The applicability of a ubiquitous language could be the entire enterprise. But often that is too big a context, so it's just for the portion of the enterprise. (Domain Driven Design refers to this as the bounded context).

Here's an example of an overlapping domain (Figure 24.1). The overlapping domain – Rental Fees and Charges – requires that the customers for both systems that interface with it agree on a common language for all things related to Rental Fees and Charges. The terms used for the rest of the Check-Out/In System and the Accounting System do not have to match. However unless the overall context is really large, it is highly preferable that they are kept matching. That way, developers and testers do not have to switch terminology meanings when they work on another system.

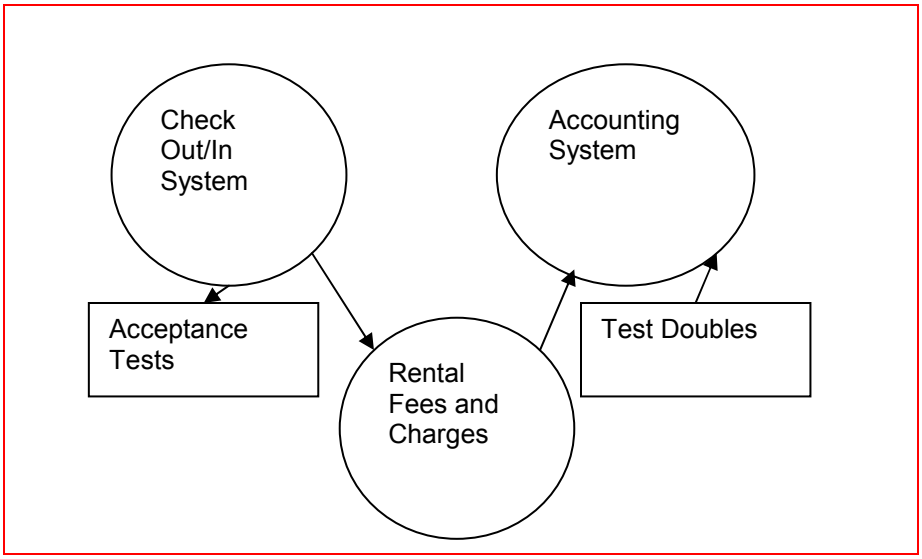


Figure 24.1 *A Common Domain – One-Way Interface*

The existence of a common domain impacts the acceptance tests. In this diagram, the interface between Check-Out/In and Fees and Charges is one-way (write-only). The acceptance tests ensure that the proper data is being transferred. The interface between Fees and Charges and the Accounting System is one-way (read-only). The test double allows the Accounting System to be tested separate from the Check-Out/In system.

Here is another common domain – CD Data (Figure 24.2). Both of the using systems (Check-Out/In and Inventory Maintenance) have a two-way interface (reading and writing) to CD Data. Since it is two way, you can use the acceptance tests for the using systems to see that they communicate with CD Data properly. You could also set up developer acceptance tests for CD Data itself.

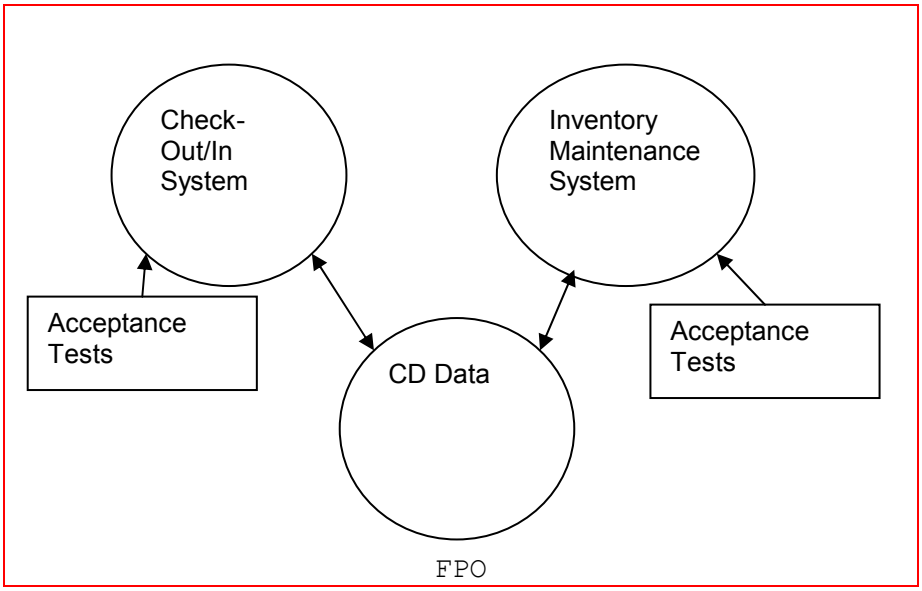


Figure 24.2 *A Common Domain – Two Way Interface*

What is a Flight?

An airline system is really large. There are reservation systems, ground operations system, and flight operations, to name a few. The term "flight" is pretty common. Consider how you, the customer, think of a flight. You are going to catch a flight. Do you use the same reference whether your journey is going to be one a single plane or whether you have to transfer between planes?

The airplane you board is for a particular flight, identified by the flight number. A particular flight number can represent the airplane traveling to many cities, only one of which you are interested in going to. Flight 1000 may go to Boston, then Philadelphia, then Raleigh-Durham, then Atlanta. However you may only be interested in the Boston to Philadelphia portion.

For some airline systems, a flight represents the entire travel of the airplane through its entire day. It includes multiple takeoffs and landings. There is an additional term for the portion between a single takeoff and the subsequent landing. That is called a leg. When airline systems communicate with each other, they can communicate about this common entity. For example, a leg is Flight 100 from Boston to Phildelphia. Each system may use the term flight in its own domain context.

As an example of continuity, the association between the airplane entity and the flight entity is not persistent. You may fly on the same flight number multiple times and get a different airplane.

Summary

- From collaboration on acceptance tests, a ubiquitous language emerges.
- Tests should be written using the ubiquitous language.
- Multiple systems using a common system need to agree on a ubiquitous language for that common system.

Chapter 25

Other Issues

"This sentence contradicts itself — or rather — well, no, actually it doesn't!"

Douglas Hofstadter

We're done with acceptance testing. No, not really. Here are a few more issues that didn't seem to fit in the other chapters.
(*** Check ordering here ***)

Context

When developing Sam's system, the requirements and tests for the process were created first – e.g. check-out and check-in. The existence of CDs and Customers were a "Given". Those tests form a context for other requirements and tests. Since entities such as CDs and Customers are needed, then there is an implicit requirement for a means to get them into the system[24].

The existence of those entities implies that we probably need Create-Read-Update-Delete functionality for each of them.[25] Create is how the entity becomes persistent. Read allows the entity's values to be read and used for calculations or displays. The need for the ability to read is usually pretty obvious. Update permits the entity's values to change. Delete removes the entity from the system. Delete can be rather complicated. For example if a customer is deleted while they have a rental outstanding, then the rental will refer to a non-existent customer. So often deletion means deactivating the entity somehow, so that it cannot be used in the future, but it persists for previously created relationships with it.

Acceptance tests can be created for the Create-Read-Update-Delete functionality for each entity. Since the form of these tests is often fairly consistent, they may be created by just the developer or tester and reviewed by the other. However, there may be a higher level of business rules that need to be applied to the operations. For example, the customer may wish to limit access to the operations. The Create-Read-Update-Delete functionality gives the context for these limitations. A table can show the permissions for every user type. For example:

Operation Security - CD				
User type	**Create**	**Read**	**Update**	**Delete**

[24] This is similar to Alexander's design by context [Alexander01]
*** Add to references [Alexander01] A Pattern Language: Towns, Buildings, Construction by Christopher Alexander, Sara Ishikawa, and Murray Silverstein

[25] Often abbreviated as CRUD.

Counter Clerk	No	Yes	Yes	No
Inventory Maintainer	Yes	Yes	Yes	Yes
Customer	No	Yes	No	No

The reverse approach would be to create an entity, as a CD; determine its security requirements; then do the Create-Read-Update-Delete functionality. Finally try to figure out what to do with all the entities. In some instances, this reverse approach may work.

Customer Examples

All examples that a customer creates can become tests. [Marick01]. But not all tests are created directly from examples. Some, like the previous entity tests are indirectly developed. If the number of tests starts to cause confusion, you can separate test cases into those that come from customer provided examples and those created by the tester or developer to check other states or calculations.

Fuzzy Acceptance Tests

Suppose Sam wants to suggest CD's that a user might rent. This could be both a user benefit and a sales tool. It's unclear exactly what CD's should be suggested. This is the case of a story that does not have absolute results. It's a fuzzy thing.

You can come up with algorithms for determining CDs to suggest. You can test the implementation of these algorithms to see that they come up with the expected results.

However it's difficult, if not impossible, to determine whether the resulting suggestions help meet Sam's expectations of more sales. Whether a suggestion grabs a customer enough for them to rent the CD cannot be measured with the types of acceptance tests we've been describing. It's too fuzzy to test. Another means for measuring the results is needed. [26]

Acceptance Test Detail

A customer such as Sam may know what he wants, but not be able to specify all the detail. For example, he might want the check-out to be faster than his competitors. But he doesn't have that detail. You can still specify acceptance tests without all the detail. It can be filled in later. Or you can use comparative results as shown in Chapter 17 as the expected outcome.

User Interface First

Debbie and Tom had Cathy the customer who could understand how the system was going to work without seeing a user interface. But what if you don't? You have a customer who needs to see a user interface in order to visualize the flow. Then create a simple interface. Demonstrate the flow of the system with as few business rules as possible and no database storage. Use test doubles as shown in Chapter 11 for simulating any actions or data that is required. The user interface prototype is a means for understanding the customer's requirements.

Once the prototype flow is approved, you can employ it as the basis for mid-tier tests. Create a story map from the flow for organization. For each display screen, assign every entry field a label. Make up action tables with those labels. Add "given" and "then" parts to the tests that give the conditions and the

[26] The sequel to TV serie "Lost" may address this.

expected results. Then make up tables that show the business rules that apply to combinations of inputs, such as the ones that have been presented in this book. The tables will clarify the cases that may have been missed by just looking at the user interface.

You could use a conceptual flow to show the steps in a workflow. (Figure 25.1) The steps could be shown with a graphic that looks like state diagram. The solid circles represent pointers to the first dialog and the exit from the last dialog. For example, the conceptual flow for check-out might look like:

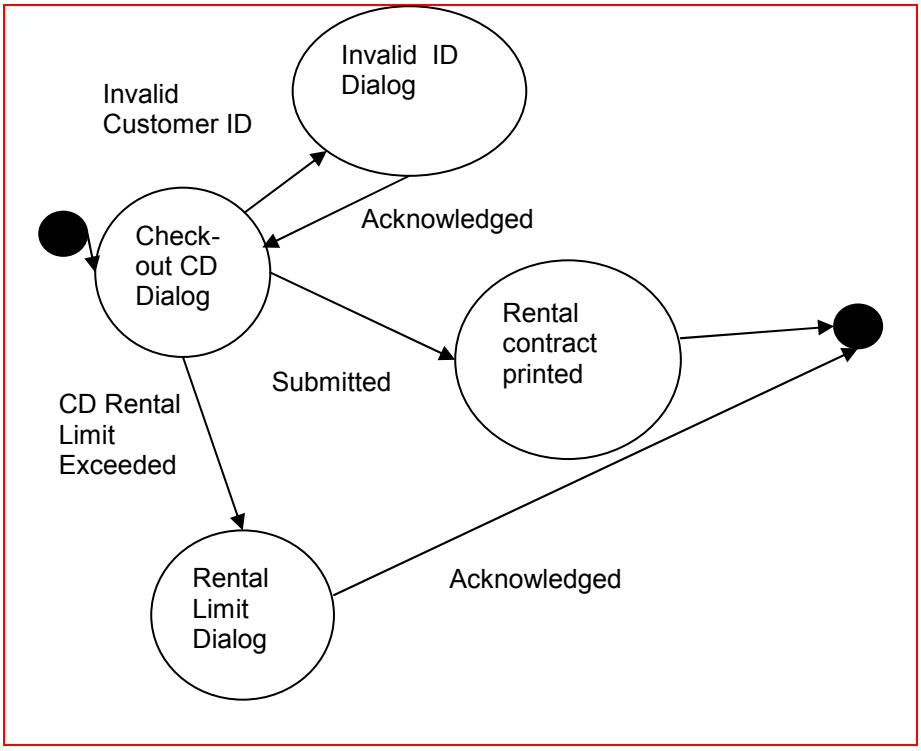


Figure 25. 1 *Conceptual Flow*

Requirements and Acceptance Tests

According to the Institute for Electrical and Electronic Engineers a requirement is a "condition or capability needed by a user to solve problem or achieve objective" or a "condition or capability met or possessed by system component to satisfy a contract, standard, specification or regulation." A requirements document usually should not include ways to implement the requirements nor a specific manifestation. Requirements should be testable/verifiable, modifiable, and prioritized. They should not be unclear, ambiguous, or incomplete (at the time of implementation). Creating acceptance tests aids in creating requirements that meet these characteristics.

The "manifestation" is how a system appears either externally or internally. It is a result of design and implementation. Requirements may include constraints on manifestation. Examples of external constraints are "The user interface shall follow the corporate web standard" or "The interface to the merchant bank shall follow their standards". Internal constraints may include specific implementation or design manifestations, such as "code in Java J2EE", "use JavaSever Faces", or "employ corporate architectural framework". Internal constraints are usually instituted to reduce the number of different

technologies that must be maintained. You may create acceptance tests for these constraints. However often the test is more a subjective measure of how well a manifestation meets the constraints.

An Anti-Missile Acceptance Test

The military likes to buy defensive weapons, such as an anti-missile missile. This has a pretty clear acceptance tests. The anti-missile missile needs to shoot down an incoming missile. If the system fails that test, it is useless.

The acceptance tests need a little more detail. Such as a defining the characteristics of the incoming missiles – speed, size, ballistic or non-ballistic, maneuvering, and so forth.

The tests need not specify the characteristics of the anti-missile missile that is needed – speed, maneuverability, antonymous versus guided. Nor do they need to measure the accuracy for the radar that tracks the incoming missiles.

All the implementation needs to do is to meet the overall acceptance test – can it destroy an incoming missile with the stated characteristics.

Documenting Requirements and Tests

You can use traditional applications for keeping requirements and associated tests, just as Word, ReqPro, or QC. Alternatively, you can keep the requirements and tests in an easily editable on-line format, such as a wiki. [27]All members of the triad can update the requirements and tests in a more collaborative environment. In either case, you should link the objectives in the charter to the features, the features to the stories, and the stories to the acceptance tests.

No Implementation Issues

The tests should not imply that you need a database underneath. You can have a slew of file clerks who took down the data, filed it away in folders, and retrieved it when requested. Of course, the system would take a little bit longer. Some people think is that because there are tables in the tests that they represent the actual database tables. Now that is one manner in how they might be implemented, but there are other approaches. They could represent views on a set of tables. Or they could be rows in a file of text. After all, they are just rows in a document.

Decoupling Requirements

Decoupling the decoupling tests from each other helps to decouple requirements from each other. This decoupling follows along the same lines as Larry Constatine's [Constatine02].decoupling of program modules that makes for higher quality code. Decoupled requirements are easier to test. There is always a possibility that requirements are somehow coupled to each other. Implementing one requirement may somehow break the implementation of another requirement unless this dependency is recognized in advance. An example of coupled requirements was given in the highly-available disk storage systems in Chapter 12.

[27] For example, Fitnesse is a wiki version of Fit created by Robert Martin and Micah Martin. . [Martin01] .

Separation of Issues

Creating customer data that matches a particular business rule can be difficult. In the initial example of the discount, there were different discounts levels based on whether a customer was Good or Excellent. There was no specification given for what determines a good or excellent customer. Suppose that some rules on past order history determined the customer rating. This might be simpler than Sam's rules in Chapter 13 for determining whether a customer could reserve or not .

Without worrying about how a Good/Excellent customer is determined, you can easily verify the business rule. You can then create Good, Excellent, and Regular customers with the history required to match the customer rating rule and run the business rule against those customers. Then you can run one or more customers through the full test (ala the first test variation in Chapter 4.

Tom might come up with other tests for the customer rating rule. For example, he might run it against the entire production set of customers to see how many customers matched each rating. That test might suggest to Betty, the business representative in Chapter 4 that a report of that type might come in handy.

A Final Set

Here is one last mix of ideas that didn't fit into other headings. .

Regression Testing

The primary purpose of acceptance tests is to translate the customer requirements into code. If you have acceptance tests for all requirements, you can use the set of acceptance tests as a regression test suite. Unless the requirement associated with an acceptance test changes, then all acceptance tests should pass. If a change is made to the implementation to accommodate a new requirement, then the previous acceptance tests should pass. [28]

Testing Systems With Random Events

Many systems have random events to which they respond. For example, the sequence of check-outs and check-ins can vary dramatically. The functionality of the response to each individual event can be tested. The correct functionality of a particular sequence of events can be tested. However, the testing of any random sequence is more difficult to test. In particular, testing an implementation where timing of the events may expose a defect is an issue in the bottom right of the testing matrix. The types of tests to uncover defects caused by errors in technical areas such as threading or locking are beyond the scope of this book.

The Power of Three

The triad represents one manifestation of the power of three. The number three occurs often in the world. Jerry Weinberg suggests that you should create at least three ways to implement a requirement so that you can make a design between them. You could explore at least three forms of tables to see which one is most suitable for the customer.

[28] "I think it's fine to add them [acceptance tests] to a regression test suite (and I would do that myself), I just don't think it should be used with automated regression testing as its primary purpose" .James Shore in an email.

The number three appears in many solutions in the non-software world. For example, there are three ways to save on transport costs if you are shipping a product. You can ship the product in larger batches; you can send it by a slower method; or you can build it closer to your customers. With at least three alternatives, you have an opportunity to compare, contrast, and pick the best or merge the three together into a better one.

How Much Developer Testing

Debbie is investigating use a bar code scanner to read the CD ID and the Customer ID. This change involves another input device, but the same functionality. There are no new acceptance tests required. Debbie needs to create developer tests for the bar code scanner. These tests ensure that the scanner can properly read the bar code. These are going to be manual tests, unless Debbie can get a robot that will move the scanner. Here are some tests she might come up with:

- Check that scanning a vendor supplied bar code produces the correct output.
- Create a bar code image with the printer, scan it, and check the output matches.
- Try scanning at different speeds and check the output
- Scan in both directions and check the output

Tom might come up with

- Try dirty bar code to see if it is readable
- Try ripped bar code to see if it is readable or gives error
- Try scanning a bar code in an off-axis direction.

All these tests need to be done with the device itself. Debbie will have one acceptance test that goes through the user interface to ensure that the connection to the scanner is properly made. But she does not have to test all the previous conditions through the user interface, unless she identifies some risk in the integration of the scanner with the system.

Cross-Application Issues

You should be able to run the tests in every environment through pre-production. In a shared testing platform or a preproduction platform, you need to ensure that the setup data for the tests is available and not stepped on by other users. If the data is only used by your application (Figure 25.2), then this should not be a problem.

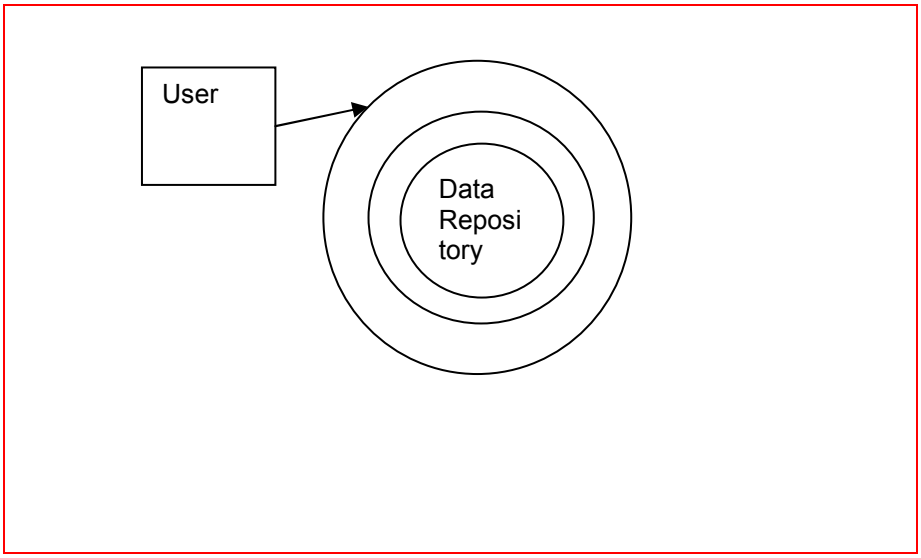


Figure 25.2 *Application Repository*

In many organizations, system testing is performed in a shared environment. This is the "play fair" environment that ensures that there are no conflicts between different programs. The persistent data in this environment is shared by many projects (Figure 25.3).

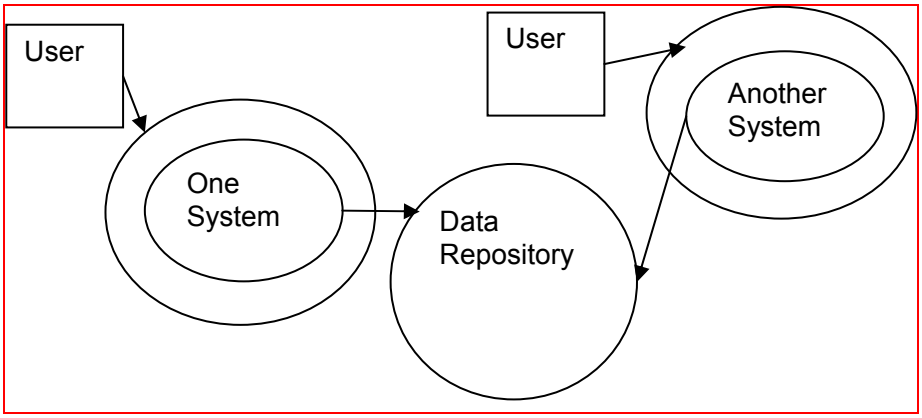


Figure 25.3 *Shared application Repository*

In this situation, tests run for one project may cause tests in another project to fail. If one project's tests add one more customer to the database, another project's test that uses the number of customers in the database may fail. These cross-project issues are often handled by each project being assigned a certain set of customers or other entities that are specific for their tests. The project tests are free to alter the characteristics of the customers, delete them, re-add them, or whatever else is necessary. The database architect can provide a custom module that restores the set of customers needed by a particular project.

Summary

- Process requirements and tests form a context for entity requirements and tests
- Specific acceptance tests may not be creatable for all requirements.
- Create a means to run acceptance tests on a shared platform without cross-application conflicts.

26

Where We've Been and Where You're Going

"Life is like an onion: you peel it off one layer at a time, and sometimes you weep."

Carl Sandburg

"There are those who look at things the way they are, and ask why... I dream of things that never were, and ask why not?"

Robert Kennedy

We've looked at acceptance tests from many different viewpoints. Here is a recap of some of the salient points.

A Recap

Tom presented an outline of the different types of testing in Chapter 3 Now that the details of acceptance testing have been explored, it's time for a recap with those details put into the overall scheme, as well as a few additional details.

The Process

The project started with a charter that gave acceptance tests for the whole project. Features with acceptance criteria were developed. The features gave an overall picture of what the system was going to do. A high-level design of the system could be created. Then the features were broken into stories, each with its own acceptance criteria. The stories were detailed in use cases or alternatively in event/response tables. Specific acceptance tests were written for the scenarios in the use cases, for individual events, or individual state transitions. Workflow tests were created which exercised more than a single use case.

Testing Layers

Many of the acceptance tests that the triad created can be run at multiple levels. For example, the check-out and check-in tests could be run as follows:.

- They can be run with real databases in an almost full integration test.
- They can be run with an in-memory database to run faster as a partial integration test.
- They can be run as a full integration tests through the user interface. The rental contract values are checked on the printer output and the credit charge output is checked on the bank statement.

- The tests through the user interface could be run manually to check for usability.

As another example, an "almost complete" integration tests might use a mock mail server which it sends mail to. A complete integration test involves using a real mail server (and appropriate email addresses so that the tests aren't a source of spam). Acceptance tests involve the entire system. After all, if the customer really wants to send an email, then the email needs to be sent to demonstrate a full end-to-end flow.

The information in the acceptance tests provides information to all the levels in a project. The flow associated with the tests and the actions performed give a framework which the user interface developers can employ in designing the user experience. The data in the tests provides the data to create the unit tests that developers use to help design and check their internal code. The database developers can design the database structure based upon the relationships between data as shown in rows and columns of the tests.

The Tests

Acceptance tests are customer-understood tests. They come from user stories, user cases, or business rules. They exercise different scenarios in use cases – the happy path and every exception path. A passing acceptance test is a specification of how the system works. A failing acceptance test is a requirement that the system has not yet implemented. Initially, all acceptance tests for a new system should fail. Otherwise, the system is already doing what has been requested. An acceptance test failure may not help in diagnose what is the problem. It simply indicates that there is a problem. Acceptance tests can be run on every platform from the developer's to the pre-production. Some may even be run on the production platform.

Unit tests are employed by the developers to help maintain and design the implementation. A developer can use the acceptance tests as a basis for developing unit tests. If there is a test at the outer layer, some module inside must help to pass that test. The unit tests can help diagnose where the problem exists that causes a failing acceptance test. There will be some duplication between acceptance tests and unit tests. After all, the unit tests may represent one case in an acceptance test. But the duplication should be minimized.

A component or module tests are architect or developer created. They work as internal acceptance tests to ensure that the individual pieces of a system correctly perform their responsibilities.

Acceptance tests suggest ways that a system might be controlled or observed at levels lower than the user interface. For example, a business rule test may connect directly with a module that implements it. A data lookup test may go to a data access layer to retrieve information to ensure that the data has been properly stored. Many tests are round trip on the same layer. For example, they perform actions on the mid-tier and check the results at the mid-tier level. Other acceptance tests may cross layers, such as input through the user interface layer and output checking from the data access layer. Meeting the needs of acceptance tests for inputs and outputs as part of the implementation will make the system more testable. The proof of testability is that a system can be tested. If the system provides a way to run the acceptance tests, then the system is testable.

Implementing an acceptance test may require additional output that is not part of the set of requirements. For example, a "CD Status Report" might show the status of every CD (Rented or Not Rented). Such a report could be used to v the results of a check-out or check-in test. But the same report might be useful in normal operations. Bret Pettichord calls the additional control points and reporting points as touch-points in the code. [Pettichord01]

Communication

There are two points to remember about communication:

- Acceptance tests and tables are not a substitute for interactive communication.
- They can provide focus for that communication.

Where's The Block?

Each member of the triad has read this book on acceptance test driven development. Okay, one hasn't'. That's a block. You need to be on the same page in order to collaborate. Just one party such as the developer raving about how wonderful ATDD may not help with the change. All members of the triad have to realize the benefits of ATDD and be in a position to implement it.

Often customers feel they do not have the knowledge to give the specifics necessary for the tests. They do not have to have all the information. The people who have it need to be identified and brought into the collaboration process. Customers may not have the time or they are not used to working at the level of precision that acceptance tests require. Once again, they need to identify someone – such as a business analyst – who has the time and can work at that level of precision. Just because the triad includes the customer doesn't mean the customer has to be there - their designated representative can fill their shoes.

In some organizations, testers, particularly ones in a separate testing group, receive an application after it has been implemented. They test the program to see if it implements the requirements as they understand them. The process would benefit where testers become part of the development team, rather than an afterthought.

Monad

Are you a monad? You get requirements without tests. You have no communication with the customer. You have no tester to help you. So write the acceptance tests anyway from what you understand the requirements to be. If you have no acceptance tests, you have no requirements. So why are you writing any code if there are no requirements?

Unavailable Customer

The customer says, "Go away and work. I've given you all the information." Now if it's an internal customer, you can appeal to shareholder fiduciary responsibility. You waste shareholder resources if you create a program that does not provide business value or one that does not meet the real needs of the customer. Don't work on that project until the customer provides specific acceptance tests. Work on another project or spend some time learning a new skill or refactoring some other application. That will be much more valuable to the company.

If it's an external customer, then be sure you are on a time and materials basis, not fixed price. You will make bundles of money in re-working the implementation of unclear requirements. Of course, the lawyers may get a piece of the action, as well as your having an unsatisfied customer who never comes back.

Change

Virginia Satir [Saiir01] developed a change model that describes how people adapt to change. When a foreign element,, such as ATDD, is introduced into an organization, it upsets the status quo, which may cause chaos. Chaos comes from the change in peoples' roles. The customer unit is more involved with providing examples. Testers create tests for an application that has not yet been written instead of seeing a user interface. Developers test more.

Exiting the chaos comes from a transforming idea - "a sudden awareness of and understanding of new possibilities". In this book, you've seen examples of the possibilities for ATDD. When you integrated the practice of ATDD into your process, you will be in a new status quo - a more effective software development organization.

Recap - Why ATDD?

The tale in this book of Debbie, Tom, and Cathy gave a narrative with the goals and benefits of ATDD woven in. A recap of these ideas seems in order. ATTD is a communication tool between the customer, developer, and tester. It is about writing the right code, rather than writing the code right. The primary goals are:

- Express details of requirements
- Discover contradictions between requirements early
- Feedback on quality
- Discovery of gaps in requirements
- Self-verifying record of business/development understanding

The secondary goals are:

- Executable regression test
- Measure progress towards "done"
- Measure complexity of requirements
- A basis for user documentation

Results

A common issue about acceptance testing is “It’s too expensive”. If you are going to test something and document those tests, it costs no more to document the tests up front than it does to document them at the end. Adding automation to the tests up-front does add a little bit of work, but it can pay off in reduced time for changes.

What have been the results of acceptance test driven development? This is what people have experienced by using ATDD. (Your Mileage May Vary). If you have issues in these areas, then ATDD should be considered.

- Improved quality (implementation is not delivered if it does not meet the tests)
- Better understanding of the system
- Identifying unclear requirements by thinking through scenarios
- Documentation of the specification

- Reduced risk of delivering a system that does not meet requirements.
- Bugs identified earlier
- Allows more courage to change
- The tests causes the code to be more testable
- Test-writing acts as a modeling process
- Eliminates translation errors between requirements and implementation
- Usable at multiple layers – user interface, mid-tier, and persistence.
- Used to help create unit tests
- Tests help form a ubiquitous language
- Less rework
- Gives programmers more knowledge of the system
- Forces conversation earlier
- Getting the detail at the necessary level
- Developers focus on what needs to be done to pass test
- Write the code that needs to be written. (If code is not executed by a test, then why is the code there?)
- Know what is going to be on the test makes it easy to code to the test
- Greater chance of covering all the cases
- Avoid last minute surprises
- Acceptance tests created prior to or during iteration planning are useful for the estimation process
- If doing waterfall, provides opportunity to discuss acceptance criteria up front.

If tests are automated, then you get the additional results of:

- Changes come faster
- Prevent bugs from creeping in from changes

What will it take?

I was teaching a course a few years ago on being lean and agile. I always ask why the students are in the course and what their backgrounds are. One student said that they had completed an agile project. The customer was more satisfied than with previous waterfall projects. The project was completed under budget and in less time. He wanted more ammunition he could present to management as to why they should do another agile project. I said that the points he raised were the best shot. I could give confirming information, but if someone isn't satisfied with those results, I'm not sure I could convince them otherwise.

-

Summary

- Acceptance test driven development makes a system testable which drives it to be higher quality.
- The bottom line is that acceptance test driven development provides benefits to an organization.

-4

Part Three – Case Studies

This part contains case studies of real-life projects.

Case Study - 1

Retirement Contributions

"First thing that I ask a new client is "Have you been saving up for a rainy day? Guess what? It's raining."

Marty, *Primal Fear*

This case study presents testing a batch process with lots of exceptions and states.

Context

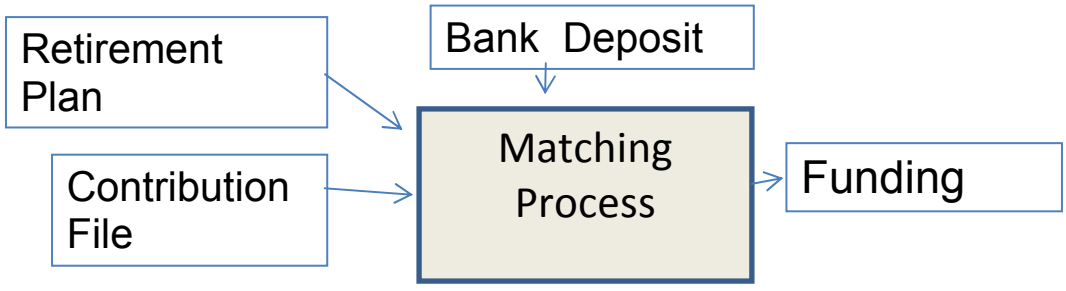
You probably have a retirement account with your company. The retirement account is administered by a financial institution that receives money from your company each month and purchases the mutual funds or other investments that you specified for your account. The financial institution receives a contribution file from your company that has a list of retirement plan participants, and the amount that should be added to each participant's account. When the institution receives a statement from its bank that your company has sent a deposit that matches the total in the contribution file, it purchases the funds for each participant.

Many retirement administrators have dedicated staffs (such as Customer Service Representatives) who match up each contribution file with the bank deposits. When a match is found, the Customer Service Representative initiates the fund purchases. Several administrators have initiated projects to automatically perform the matching operation.

There are many issues that can occur with the matching process. The amount of the deposit may not exactly match the total of amounts in the contribution file. The deposit notification may not arrive for several days after the contribution file is received. There may be multiple contribution files which are matched by a single deposit.

The business case for these projects is to cut down on the time spent by the Customer Service Representatives.

Here's the overall flow of the process. The matching process checks the participant identifiers in the contribution file against the corresponding retirement plan. If each identifier matches and the bank deposit is correct, then the funding data is produced.



The Main Course Test

The main course test assumes that the deposit amount exactly matches the total on the amounts in the contribution file and that all identifiers listed in the contribution file correspond to participants in the plan.

Setup

Each retirement plan has a set of participants who have decided how to invest their contributions. Each plan is associated with a bank account from which transfers are made to pay for purchasing of the funds.

Retirement Plan	**Plan ID = XYZ**	
Name	**Participant ID**	**Fund**
George	111111111	Wild Eyed Stocks
Sam	222222222	Government Bonds
Bill	333333333	Under The Mattress

Banking Relationship		
Plan ID	**Bank Routing Number**	**Account Number**
XYZ	555555555	12345678

Event

The event consists of two parts, which can occur in either order. One is the arrival of the contribution file and the other is the arrival of the deposit notification. The Date Time Set makes the example repeatable.

Date Time Set
January 30, 2010 08:18 AM

Here is the contribution file where all participant IDs match those in the plan.

Contribution File	**Plan ID = XYZ, File ID = 7777**
Participant ID	**Amount**
111111111	5000
222222222	1000
333333333	500

Here is the Bank Deposit whose amount matches exactly the total of the amounts in the Contribution File.

Bank Deposit			
Routing Number	**Account Number**	**Amount**	**Deposit ID**
555555555	12345678	6500	8888

Expected

Since the conditions for matching have been met, the expected output is funding instructions

Funding	**Plan ID = XYZ, File ID = 9999**	
Participant ID	**Fund**	**Amount**
111111111	Wild Eyed Stocks	5000
222222222	Government Bonds	1000
333333333	Under The Mattress	500

Matches			
Deposit ID	**Contribution File ID**	**Funding File ID**	**Matched On Date Time**
8888	7777	9999	January 30, 2010 08:18 AM

Implementation Issues

How the setup works depends on your particular testing environment. The setup tables Retirement Plan and Banking Relationship can either simply check that the corresponding entries exists in the appropriate databases or insert them into the databases if they do not exist. If they do not exist and they cannot be inserted, the test fails at setup.

The contribution table and the bank deposit table can be converted by the fixture into data files that match the format of the data received by the system from the companies and the bank. These data files can then be processed by the matching program as if they were actual files.

The matching program produces a funding file that is processed by another system. This funding file can be parsed and matched against the expected funding output.

Business Value Tracking

Manual matching of deposits and contribution files takes an extensive amount of Customer Service Representative's time. In a majority of the instances, the match could be processed automatically. If the conditions did not hold (exact match of amounts and all participants already enrolled in the plan), the matching process could be performed with the current manual process. So there is high business value in creating a program that handles the main course as quickly as possible. The user story that encompasses doing the main course is given a large business value.

There can be a number of exceptions during the process. The deposit and contribution total may be off by a small amount (e.g. a few cents) or a large amount. There may be a participant listed in the contribution file that is not entered as a participant on the plan. In that case, the matching program does not know what fund should be purchased for that participant.

Separation of Concerns

There are many other problems that can occur during the entire operation of the system. The received contribution file may not be in a readable format. This may occur the first time the file is received from a company due to setup issues, or it may occur repeatedly because of ongoing issues at the sending

company. The files may be in different formats due to each individual company's human resource system. Those problems can be dealt with by using another set of tests that read samples of actual input files and checks that either the file can be translated into a common format or indicates a conversion error that must be dealt with manually.

First Exception

Each exception should have its own test to show that the exception is handled properly. The tests may use a common setup (See Chapter 20.)

Event

The setup is as for the main course. The only difference in the event is that the deposit is off by one cent. The customer unit decided that any discrepancy less than a dollar should be handled by the equivalent of "take a penny, give a penny". The discrepancies will be kept on some form of persistent storage (a database table or a log file). At some point they can be analyzed in order to determine if there is some systemic issue such as one company always being two cents short. For the time being, the total of the discrepancies will be reportable to the appropriate financial officer so that the books can be balanced.

.

Date Time Set
January 30, 2010 08:18 AM

Contribution File	**Plan ID = XYZ, File ID = 7778**
Participant ID	**Amount**
111111111	5000
222222222	1000
333333333	500

Bank Deposit			
Routing Number	Account Number	Amount	Deposit ID
555555555	12345678	6499.99	8889

Expected

Since the conditions for matching have been met, the expected output is funding instructions

Funding	**Plan ID = XYZ, File ID = 10000**	
Participant ID	**Fund**	**Amount**
111111111	Wild Eyed Stocks	5000
222222222	Government Bonds	1000
333333333	Under The Mattress	500

Matches			
Deposit ID	**Contribution File ID**	**Funding File ID**	**Matched On Date Time**
88889	7779	10000	January 30, 2010 08:18 AM

Now the issue is whether to show this as a separate table or should it be part of the Matches table. Since purpose is for balancing non-matching funding, it is shown as a separate table.

Discrepancy			
Deposit ID	**Contribution File ID**	**Matched On Date Time**	**Discrepancy Amount**
88889	7779	January 30, 2010 08:18 AM	-.01

Another Exception

Once again, a common setup could be used. In this exception, a participant id that is listed in the contribution file does not have a corresponding entry in the retirement plan.

Event

The setup is as for the main course. The difference is that the contribution file contains an additional entry. The total matches the deposit.

.

Date Time Set
January 30, 2010 08:18 AM

Contribution File	Plan ID = XYZ, File ID = 7779
Participant ID	**Amount**
111111111	5000
222222222	1000
333333333	500
444444444	100

Bank Deposit			
Routing Number	**Account Number**	**Amount**	**Deposit ID**
555555555	12345678	6600	8890

Expected

Since the conditions for matching have been met, the expected output is funding instructions.

Funding	Plan ID = XYZ, File ID = 10001	
Participant ID	**Fund**	**Amount**
111111111	Wild Eyed Stocks	5000
222222222	Government Bonds	1000
333333333	Under The Mattress	500

Matches			
Deposit ID	**Contribution File ID**	**Funding File ID**	**Matched On Date Time**
88890	7779	10001	January 30, 2010 08:18 AM

The missing participant needs to be reported somehow. A separate table shows the information that is attributed to the participant. A separate user story and set of tests will show how to handle this output.

Missing Participant			
Plan ID	**Participant ID**	**Matched On Date Time**	**Participant Amount**
XYZ	444444444	January 30, 2010 08:18 AM	100

Two Simultaneous Exceptions

So what if two exceptions both occur in the same processing? Should there be a test for that? Often it is difficult to form an automatic response to the occurrence of two exceptions for the same transaction. So the transaction is handled manually. The test then should show that an exception occurred. If the two exceptions are decoupled (the response to one does not depend on the response to the other), then the tests for the individual exceptions may be sufficient, depending on the risk tolerance of the project.

Event

Once again, the setup is as for the main course. But both a missing participant and a non-matching deposit are involved.

.

Date Time Set
January 30, 2010 08:18 AM

Contribution File	Plan ID = XYZ, File ID = 7780
Participant ID	**Amount**
111111111	5000
222222222	1000
333333333	500
444444444	100

Bank Deposit			
Routing Number	**Account Number**	**Amount**	**Deposit ID**
555555555	12345678	6599.99	8891

Expected

No funding is produced. The output describes the exceptions that occurred during the matching process. These exceptions can be tracked to determine which combinations of multiple exceptions frequently occur. At some point, the frequent combinations could be handled in code, rather than left for manual processing.

Exception		
Contribution File ID	**Deposit ID**	**Exceptions**
7780	8891	Deposit_does_not_match_contribution_total Participant_in_contribution_file_not_in_plan

The Big Picture

These tests have been focused on the context of the system. In the big picture, not only does a funding file have to be created, but the funds must actually be purchased and the transactions need to be recorded in each participant's account. A larger test for this entire workflow needs to be developed. It may not necessarily be run as an automated test. It may still require some mock implementations. You would not want to keep purchasing mutual funds and adding them to a participant's account every time you run this large test. So some mock for the actual purchasing interface would be required.

The big picture test is beyond the developer unit's scope. Their job is to ensure that the funding instructions are correct, based on the input. But the project is not complete until the full test is run by the testing unit.

Event Table

The matching process is a batch process. It is not driven by user input, but by events that occur. So an event table is quite appropriate for this case. Here's an example:

Matching Events		
Event	**Response**	**Notes**
Contribution File Received	Check for matching deposit If so, perform match Else store file	
Bank Deposit Received	Check for matching contribution file	

	If so, perform match Else store deposit	
One week after Contribution File Received	If no bank deposit received, notify client	
One week after Bank Deposit Received	If no contribution file received, notify client	
One minute prior to market close	Disallow matching until after market close	Prevents funding issues

The events are those that are external to the system.

State Table

The Contribution File goes through several states. These states keep track of its progress through the matching process. These are some of the states that a file can be in:

Contribution File State	**Meaning**
Received	Contribution File received
Data Checked	File has been examined for format errors
Awaiting Match	Waiting for Bank Deposit
Perform Match	Bank Deposit has been received
Check Match	Bank Deposit / Contribution File match has been attempted
Purchase Funds	Matched Bank Deposit, buy funds
Begin Exception Processing	Bank Deposit received, but has problems
Edit Processing	Contribution File has bad format

The external events and internal events cause the state of the contribution file to change or some processing to occur. This table describes some of the state transitions for the contribution file.

Contribution File State / Event Transitions				
State	**Event**	**Response**	**New State**	**Notes**
None	Received contribution file	Record contribution file	Received	
Received		Perform data check	Data Checked	Examine file for format errors
Data Checked	Data check is bad		Edit Processing	
Data Checked	Data check is good		Awaiting Match	
Awaiting Match	Bank deposit received		Perform Match	
Perform Match		Process Match	Check Match	
Check Match	Matched okay		Purchase Funds	

Check Match	Discrepancy in match		Begin Exception Processing	
Awaiting Match	Matching time exceeded		Begin Exception Processing	One week after receipt
Purchase Funds	Time greater than one minute prior to market close	Enter order for funds	Funds Purchased	
Purchase Funds	Time less than one minute prior to market close	Put onto purchase for tomorrow	Funds Awaiting Purchase	
Begin Exception Processing		Put on Customer Representative's Work Queue		

There can be a test associated with each state transition that is not already being tested by another test. For example, for the first transition, you might have:

Given:

Retirement Plan	**Plan ID = XYZ**	
Name	**Participant ID**	**Fund**
George	111111111	Wild Eyed Stocks
Sam	222222222	Government Bonds
Bill	333333333	Under The Mattress

When a contribution file is received

Date Time Set
January 30, 2010 08:18 AM

Contribution File	**Plan ID = XYZ, File ID = 7777**
Participant ID	**Amount**
111111111	5000
222222222	1000
333333333	500

Then record it as it is received.

Contribution File States		
Plan ID	**File ID**	**Status**
XYZ	7777	Received

Summary

- Separate concerns to make for easier testing
- Each exception should have its own test
- Test every state transition

Case Study - 2

Email Addresses

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Knuth

This study involves breaking down a complex test, which represents a complex business rule, into simpler tests. The simpler tests can decrease time to understand and implement.

Context

Almost every application that involves communication today requires email addresses. When an email address is entered, it should be validated to ensure that it is in a valid format. The process of whether an email address is actually valid requires an exterior action, such as sending an email to the address and checking that it was not rejected.

The testers for one company had a set of email address examples they would use to test every application that had an email address entry. The examples included both correct and incorrect formatted addresses. The company also had a business rule that email addresses from some domains were not acceptable. These domains included mail servers that allowed completely anonymous email. A portion of the tests are shown here:

Email Tests		
Email	**Valid?**	**Reason**
George@sam.com	Y	
George@george@same.com	N	Two @
.George@sam.com	N	Invalid name
George@samcom	N	Invalid domain
George+Bill@sam.com	Y	+ is allowed in name
George@hotmail.com	N	Banned domain
George@iamoutogetyou.com	N	Banned domain
… and many more		

Every time an email address was entered on an input screen, this entire set of tests was run.

An Email Trick

You may notice George+Bill@sam.com is a valid email address. The '+' sign is allowed in the part prior to the @ sign. It has a special meaning in most email systems. The email is delivered to the part before the '+' sign ("George" in this case). The full address ("George+Bill") is used in the "To" part of the email. Your email client (Eudora, Thunderbird, Outlook, etc.) can filter based on the part after the '+' ("Bill") to put the message in a particular folder or perform another action. Some email systems or internal servers may discard the '+' so that the mail goes to GeorgeBill@sam.com. That represents a different user than George. So you should test this on every system to ensure that it works.

Breaking Down the Tests

This acceptance test is pretty clear. When I ask developers how much time they think it will take to program an email validation routine that will pass this test, they suggest two days or so.

Let's break down the test into smaller tables. We saw in Chapter 13 that breaking down a business rule into simpler business rules makes things easier to understand and test. Each smaller table represents either a requirement or a test. Sometimes it's hard to distinguish between the two. The tables presented here represent either details of an email address from the business point of view or unit tests for a module that implements email format verification.

In the email situation, we can use the rules that the internet standards provide. In RFC 2822 [IEFF01] The first rule for a valid email address is that it contain one and only one '@' symbol. The '@' separates the name part of the address from the domain part. For example with "ken.pugh@netobjectives.com", "ken.pugh" is the name part and "netobjectives.com" is the domain. The official term for the name part is "local-part". So we will use that in our tables.

Email Split Into Local-part and domain			
Email	**Valid?**	**Local-part?**	**Domain?**
X@Y	Y	X	Y
XY	N	DNC	DNC
XY@XY@XY	N	DNC	DNC

Usually a developer says that it will take less than fifteen minutes to implement code that performs this check. It all depends how much they are familiar with the regular expression library.

Local-part validation

According to the rules, the local-part must only contain the characters:

- Uppercase and lowercase English letters (a-z, A-Z)
- Digits 0 through 9
- Period (.) provided that it is not the first nor last character,
- nor may it appear two or more times consecutively.

- Characters ! # $ % & ' * + - / = ? ^ _ ` { | } ~

The maximum size of the local-part is 64 characters.[29] We can put this free text business rule into a table:

Local-part Allowable Characters		
Characters	**Allowed?**	**Notes**
a-z	Y	
A-Z	Y	
0-9	Y	
! # $ % & ' * + - / = ? ^ _ ` { | } ~	Y	
.	Y	If not first, last or two consecutive
Anything else	N	

We could have a test of the allowed characters for the local-part. Of course we need some test cases to see that the character rules are applied properly.

Local-part character combination tests		
Local-part	**Valid?**	**Notes**
George	Y	
George^	N	Character not allowed
George..a	N	Period appears twice in a row
.George	N	Period first
George.	N	Period last

And of course, we should have some rules for length.

Local-part length tests		
Local-part	**Valid?**	**Notes**
	No	Zero length
a	Yes	Minimum length
12345678901234567890123456789012345678901234567890123456789012345678901234	Yes	Maximum length
123456789012345678901234567890123456789012345678901234567890123456789012345	No	Exceeds maximum length

[29] If you are actually following this case study to do your own validation, you may also wish to eliminate characters that can inject SQL. [Security01] You would not allow characters as a single quote or a forward slash.

Now when I ask a developer how long it would take to implement just the local part rules, they usually say no more than an hour. They usually implement these rules in two methods.

Domains

The rules for domains are different than for the local part. The allowed characters are:

Domain Allowable Characters		
Character	**Allowed?**	**Notes**
a-z	Y	
.	Y	Must have at least one Cannot be first or last character
-	Y	
0-9	Y	
Anything else	N	

A domain has a maximum of 255 characters. Periods separate the parts of a domain. The basic structure is that the top level is the right most part of the domain, the second level is the next most right part, and so forth. So for a domain with two periods, the levels are:

Third-level.second-level.top-level

For example, "www.netobjectives.com" has three levels. The third-level is "www"; the second-level is "netobjectives"; the top-level is "com".

The rule for levels is there must be at least a top level domain and a second level domain. Most email addresses use just two levels, but there is no limit to the levels, within the confines of the maximum of 255 characters. Top level domains are standard, such as "com", "org", "net", "us", "ca", "mx", "tv", etc.

We have a few alternatives for the top-level domain part. We could institute that the top-level part be at least two characters and at most four characters. Our application would accept top-level domains that might not be valid. For example

Top Level Domain Parts		
Top Level	**Valid?**	**Notes**
t	No	Too short
tv	Yes	Minimum length
abcd	Yes	Maximum length
abcde	No	Too long

Alternatively we could have a table that lists all valid top-levels, such as:

Top Level Domain Parts	
Top Level	**Valid?**
com	Yes
net	Yes
us	Yes

… and so forth for all possibilities	Yes
Anything not listed	No

The issue with having something that lists them all is that when a new top-level domain is created, the table will have to be updated. However, you can insure that there is validation for all the ones that are currently in the table. There is a generic aspect to this trade-off. The more specific you are, the less possibility that something invalid will sneak through. However less specific you are, then you won't have to update anything if something new that fits into the mode passed through. Your choice, based on customer input.

Here are the tests for the domain structure.

Domain Breakdown Tests					
Domain tests	**Top Level Domain**	**Second Level Domain**	**Third Level Domain**	**Valid?**	**Notes**
A.B.COM	COM	B	A	Y	
COM				N	Must have at least one period
B.COM	COM	B		Y	Could require two
A.B.C.COM	COM	C	B	Y	Fourth Level is D
A.B.COM.				N	Cannot end in period
.A.B.COM					Cannot begin with period

And of course, we can have a test for the maximum length of the domain. That is left to an exercise for the readers.

Developers, when asked, usually suggest that it would take an hour or so to code up the method that this table could be tested against.

Disallowed Domains

Now we have a list of disallowed domains. These are domains that customers cannot use. The domains permit people to send anonymous emails. Whether a particular application wants to disallow these domains is up to it. The list is static. If we wanted to be able to update the list, then we would create tests to verify that the list is updated correctly. This table represents the list. We can use it as input to the program or as a test to see that every domain on it is recognized to be a disallowed domain.

Disallowed Domains	
Domain	**Reason**
Hotmail.com	Anonymous mail
Imouttogetyou.com	Spam source
Anymore ???	

When developers are asked how long it would take (without worrying about updating the list), they usually answer around an hour.

The total time estimates from the individual pieces is usually around four hours. The original time estimate from the combined test is often two days. By breaking the tests down into smaller parts, the

estimated time goes down, since each part appears simpler. In addition, the smaller parts are easier to program.

Back To The Beginning

You can run the same tests as before against a module that performs all of these tests. That would ensure that all the validation functions have been tied together properly. It's possible that the test cases in this table that fail because the results in this table differ from the actual result. In that case you get to decide whether you should alter the underlying implementation to agree with these tests or alter these tests, because they actually do not represent email address validation correctly.

Email Tests		
Email	**Valid?**	**Reason**
George@sam.com	Y	
George@george@same.com	N	Two @
.George@sam.com	N	Invalid name
George@samcom	N	Invalid domain
George+Bill@sam.com	Y	
George@hotmail.com	N	Banned domain
George@iamoutogetyou.com	N	Banned domain
… and many more		

Is It Really Valid?

There are two parts to the validation process. The first is to make sure that the email address is properly formatted. The second is to ensure that a properly formatted email address actually represents a real email address. Usually, you send an email to the address. If it bounces, then it is invalid. The email often contains either a note to reply to the message or a link to a web page that can record that the message was received. The table for this might look like:

Actual Email Address Check			
Sent Message	**Bounced**	**Received Reply**	**Valid?**
Yes	No	Yes	Yes
Yes	Yes	Do not care	No
Yes	No	No	Unknown
No	Do not care	Do not care	Unknown

Summary

- Break up complicated conditions into smaller conditions
- Smaller conditions are easier to test and code
- Validation of correct format is only part of validation
- Validate a correctly formatted value represents a real value (e.g. email address, customer id, CD id, etc.)

Case Study - 3

Signal Processing

Lisa Simpson: Would you guys turn that down!

Homer Simpson: Sweetie, if we didn't turn it down for the cops, what chance do you have?

"The Simpsons" *Little Big Mom* (2000)

Acceptance tests for a real-time signal processing system are presented.

It's Too Loud

I have produced software with Richard Cann of Grozier Technical Systems for a number of years. Grozier produces sound level measurement systems. These systems are used by numerous concert sites, particularly outdoor ones, to monitor sound levels for regulatory reasons and to act as good neighbors.

The systems are composed of embedded systems which run multiple programs. Richard produces some of the programs, particularly the display and control version. I produce the real-time signal analysis portion. [Wiki08]

Sound Levels

The programs input the sound through the microphone input. Each second, the 22,500 samples are recorded. The signal analysis programs perform calculations on each of these sets of samples. The output is the equivalent continuous sound level (Leq). An overview of the process is shown in Figure X.1.

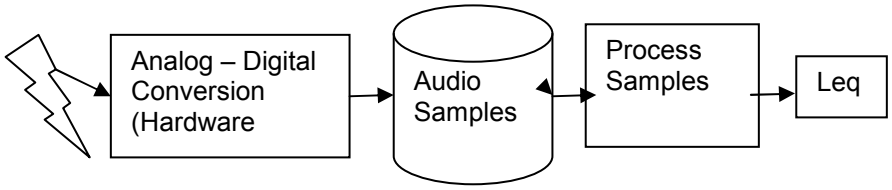


Figure X.1 Sound Level Process

The sound is input through a microphone. The gain (how much the signal is amplified) varies based on both the microphone and the volume setting on the input. To correctly compute the Leq, the gain must be adjusted so that a standard sound source (a calibration source) produces a specific Leq result.

It is difficult to replicate the entire system (microphone, calibration source, and so forth) for testing. However the sounds can be captured by regular recording methods and then replayed as input for the

tests. Suppose the sounds produced by the calibration source are captured in a calibration file. And that sounds of known Leq are also captured in separate files. Then the test of the overall system can be:

***Start Background

Given that the volume is adjusted so that calibration file (containing a 1 KHz signal) yields the standard Leq:

Calibration	
File	**Leq (db)**
"calibration.wav"	94

Then the test files should be produce values of the expected Leq.

Leq Tests	
Input File	**Leq (db) ?**
"test1.wav"	88
"test2.wav"	95

***End Background

In order to ensure that these Leq's are correct, the same test can be repeated with calibrated specialized hardware. The process of comparing two separate ways of calculating an output and comparing them is similar to comparison of the outputs the multiple flight control algorithms that the space shuttle employs. Each algorithm computes a path. If all results agree, everything is great. If one differs, the shuttle uses the results that majority agree with. If every result is different, well, that's what makes for an interesting flight – who do you believe?

Developer Tests

The details of the process are shown in figure X.2. It shows that there are two intermediate results in the computation - the windowed samples and the a-weighted samples.

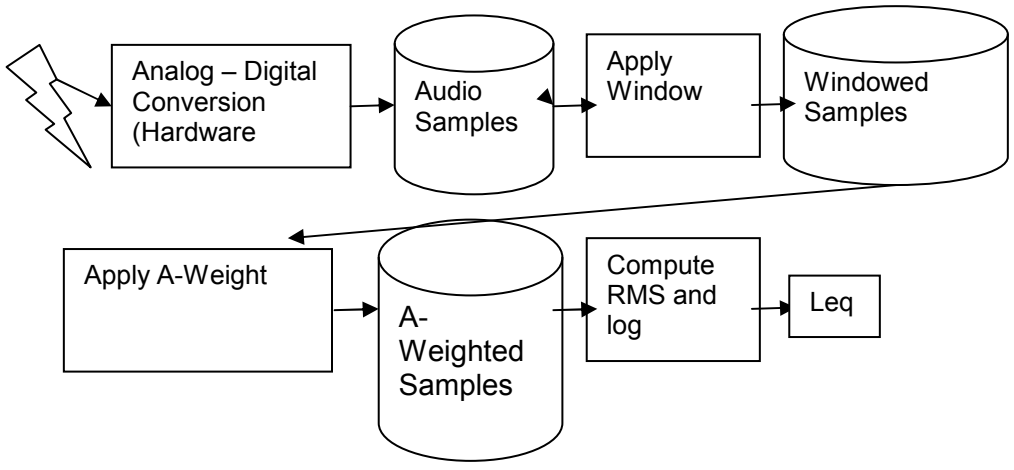


Figure X.2 Details of Sound Level Process

Richard is particularly interested in the final results – is the Leq computed correctly for a particular file? However as a developer, it helps if I can apply tests to intermediate results. These intermediate tests would usually be termed unit tests, since they apply to lower level modules. However, since Richard is highly experienced in signal processing, he creates input and output files that can be used to ensure that the intermediate processing works correctly. For example:

Compute Leq	
A-weighted sample file	**Leq?**
Aweight1.data	77
Aweight2.data	44

The input and output files contain digital samples for one second. Equivalent files are available for each of the other steps in the process.

Summary

- Acceptance tests do not have to involve just simple values. They can be entire sets of values (usually represented in files)
- A subject matter expert can often create lower level tests that can be used as unit tests
- Tests for real-time systems may run for a while

Case Study - 4

A Library Print Server

"There's no such thing as a free lunch"

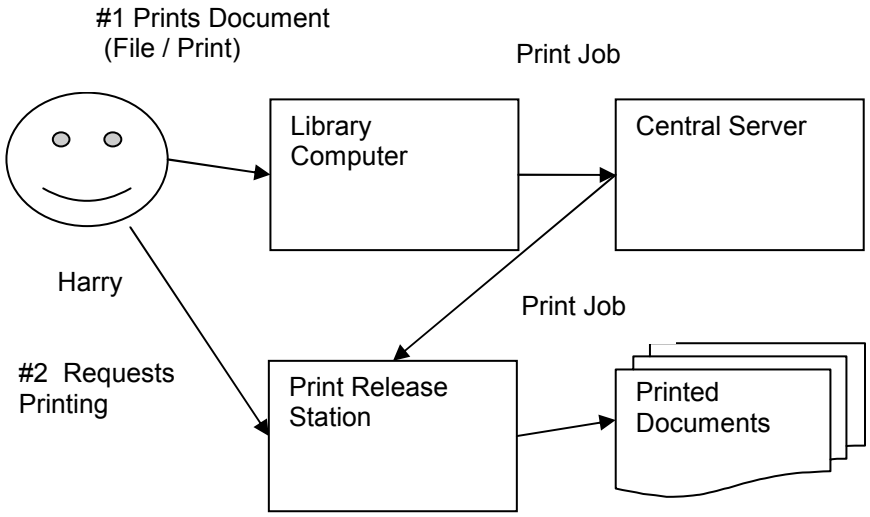
Anonymous

Here is a real-life library print server system. Libraries use such a system to charge for printouts of documents. The example shows how acceptance tests can cover a workflow and not just a use case.

The System

I had consulted for Rob Walsh, the cofounder of EnvisionWare in exploring a new object design for the print server system they provide to libraries. In my book *Prefactoring,* I showed the unit test strategy and underlying object design for the system. [30] I described the work flow with a concentration on how it was implemented using the internal messaging system. The following workflow concentrates on the acceptance tests.

A library patron, Harry, wants to print a document created on one of the library's computers. Harry submits the document for printing and then goes over to a release station to print the job. There are two separate use cases – submitting the document for printing and actually printing the document. – which form the workflow. Internally the personal computer and the release station communicate with a central server.



[30] See Chapter 15 in [Pugh01]

There are a series of simple tests that precede this workflow test. Some tests ensure that the print cost is correctly computed for various users and different modes of printing – black and white and color. Others check that the user can deposit money into their pre-paid account.

A Workflow Test

In this workflow, Harry submits two documents for printing and then goes to the release station. When Harry signs onto the release station, he sees a list of the two print jobs. Harry selects each one to print. Each job is printed and Harry's account is charged. Here's the detailed test:

***Start background

Given a user with an account on the system

User				
Name	**Balance**	**User ID**	**B&W Per Page Rate**	**Color Per Page Rate**
Harry	$1.00	123	.03	.10

And two documents to be printed

Document		
Name	**Number Pages**	**Contains**
Harrystuff	1	
The quick brown fox jumped over the lazy dogs		

Document		
Name	**Number Pages**	**Contains**
Morestuff.doc	7	
--Lots of stuff--		

And no print jobs currently on the print queue for that user

Print Jobs	**User ID = 123**		
User ID	**Filename**	**Print Mode**	**Job Number**

When the user requests the first document to be printed

Request Printing		
Enter	**User ID**	123
Enter	**Filename**	Harrystuff.doc
Enter	**Printing Mode**	Black & White
Press	**Submit**	

Then the system responds with:

Approve Print Charge		
Display	**Print Charge**	$.03
Press	**Accept**	

If the user accepts, a print job is created

Print Jobs			
User ID	**Filename**	**Print Mode**	**Job Number**
123	Harrystuff.doc	Black & White	99991

When the user requests a second document to be printed

Request Printing		
Enter	**User ID**	123
Enter	**Filename**	Morestuff.doc
Enter	**Print mode**	Color
Press	**Submit**	

Then the system responds with:

Approve Print Charge		
Display	**Print Charge**	$.70
Press	**Accept**	

If the user accepts, a print job is created

Print Jobs			
User ID	**Filename**	**Print Mode**	**Job Number**
123	Harrystuff.doc	Black & White	99991
123	Morestuff.doc	Color	99992

***End Background

Now two print jobs have been created. The next step in the flow is for Harry to go over to the release station and request each job to be printed.

***Start Background
Given that print jobs have been created for a user

Print Jobs			
User ID	**Filename**	**Print Mode**	**Job Number**
123	Harrystuff.doc	Black & White	99991
123	Morestuff.doc	Color	99992

When the user enters his user id

User ID Entry		
Enter	**User ID**	123
Press	**Submit**	

Then the list of print jobs is displayed

Print Job List	
Filename	**Job Number**
Harrystuff.doc	99991
Morestuff.doc	99992

When the user selects one file,
Then it is printed on the appropriate printer.

Printed output?	**Selected File = Harrystuff.doc**	
The quick brown fox jumped over the lazy dogs		

Then the file is eliminated from the display list and the print queue. The user's account is charged the print cost.

Print Job List	
Filename	**Job Number**
Morestuff.doc	99992

Print Jobs			
User ID	**Filename**	**Print Mode**	**Job Number**
123	Morestuff.doc	Color	99992

User		
Name	**Balance**	**User ID**
Harry	$.97	123

When the user selects another file, it is printed.

Printed output?	**Selected File = Morestuff.doc**	
--Lots of stuff--		

Then it is eliminated from the display list and the print queue. The user's account is charged.

Print Job List	
Filename	**Job Number**

Print Jobs	User ID = 123		
User ID	**Filename**	**Print Mode**	**Job Number**

User		
Name	**Balance**	**User ID**
Harry	$.27	123

*** End Background

There could be additional tests that show the flow if the user does not have sufficient money to print a document or if the user decides to cancel the printing of a document. Those tests are left as an exercise for the reader

Summary

- A workflow test consists of more than one use case

Case Study 5

Highly Available Platform

"You just call out my name,
And you know where ever I am
I'll come running, oh yeah baby
To see you again".

James Taylor, *You've Got A Friend*

Many corporate software systems depend on a highly available platform. This chapter shows some increasing detailed acceptance tests for such a platform.

Switching Servers

A highly available platform has at least two independent computers. If one goes down, the other available computers take over the load. If the servers are running close to capacity, not all the applications may be able to run. A pre-determined priority mechanism determines which applications get to run. In addition to switching applications when a server goes down, in the case study, the system administrator should be notified via either email or a text message.

The capacity of a server to run applications depends on the demands of the applications, such as memory, processor usage, and input-output operations. Instead of being overwhelmed by all the details at once, this study shows how the details can be introduced gradually. This is another manifestation of the separation of concerns guideline.

The first test uses a simplified capacity that considers just the number of applications that can be run on each server. This test demonstrates that the servers switch applications properly when one of them goes down. There will be a lot of technical work to perform in order to make that happen. So starting with this simple acceptance test keeps the developer unit busy while the customer unit examines more detailed capacity issues.

**Start Background

Given these servers

Servers	
Name	**Capacity (Applications)**
Freddy	5
Fannie	3

And these applications with their priority

Applications	
Name	**Priority**

CEO's Pet	100
MP3 Download	99
Lost Episode Watching	98
External Web	50
Internal Web	25
Payroll	10

And the servers running these applications

Server Load	
Name	**Applications**
Freddy	CEO's Pet, External Web, Internal Web, Payroll
Fannie	Lost Episode Watching, MP3 Download

When a server goes down, switch any applications running on it to the alternative server. If the alternate server does not have the capacity for all the applications, then run the applications based on priority order.

Server Events			
Event	**Servers Remaining ?**	**Applications Running ?**	**Action?**
Freddy Goes Down	Fannie	CEO's Pet, Lost Episode Watching, MP3 Download	Send event alert "Freddy down"
Fannie Goes Down	Freddy	CEO's Pet, Lost Episode Watching, MP3 Download, External Web, Internal Web,	Send event alert "Fannie down"

***End Background

A second test ensures the next part of the flow is proper. Now that an event has occurred, the administrator should be notified in the appropriate way.

*** Start Background

Given an event alert has occurred and these preferences

Administrator Notification			
Notification Preference	**Text ID**	**Email**	**Action?**
Email	123	Ab@e.com	Send mail to AB@e.com

Then send the event alert to the system administrator based on the notification preference that the event occurred. If the notification fails, send the event alert via the other method.

Notification	**Action = Send mail to AB@e.com**
Response	**Action?**

Mail sent	None
Mail not deliverable	Send text to 123

***End Background

A similar test can be created for three servers with one of them going down. That test would how the applications should be distributed among the remaining two servers. That test is left as an exercise for the reader.

More Complications

Now that the basic concept of how application switching works, more complex rules can be applied to capacity of each server. The selection of what applications to run is a separable concern. The results can be tested independently of testing the switching functionality.

If only a single server is running, the applications that can be run depend on selecting the highest priority applications that can run within the capacity. It's possible that an application that has a higher priority cannot be run because there is insufficient capacity. Then a lower priority application that does not require as much capacity might be run.

In the following table, CPU usage is measured in MIPS – millions of instructions per second. Memory usage is calculated in megabytes (MB). Input-output usage is measured in total of reads and writes per second (RWS).

In this test, "Lost Episode Watching" requires 5 MIPS. The two higher priority applications – "CEO's Pet" and "MP3 Download" use 11 MIPS of the 14 MIPS available. So it cannot be run, but a lower priority application "External Web" only requires 3 MIPS, so it can be run.

*** Start Background

Given applications with these characteristics

Application Characteristics				
Name	**Priority**	**CPU Usage (MIPS)**	**Memory Usage ()**	**Input - Output Usage (RWS)**
CEO's Pet	100	1	1000	1000
MP3 Download	99	10	500	2000
Lost Episode Watching	98	5	200	3000
External Web	50	3	400	500
Internal Web	25	1	100	500
Payroll	10	8	10	1000

And a single server with the following capacity:

Server Characteristics			
Name	**CPU Usage (MIPS)**	**Memory Usage ()**	**Input - Output Usage (RWS)**
Fanny	14	2000	5000

Then it should run the following applications:

Application Running			
Name	**CPU Usage (MIPS)**	**Memory Usage (MB)**	**Input - Output Usage (RWS)**
CEO's Pet	1	1000	1000
MP3 Download	10	500	2000
External Web	3	400	500

***End Background

Similar tests can be created for two or more servers. The creation of these tests brings up issues of how applications should be balanced between two servers. Given this test, it's clear if there were a second server, at least "Lost Episode Watching" should be running, as long as its capacity was greater than that application's needs.

The tests for what applications should be run on what servers can become fairly complicated. There are usually more issues, such as applications that need specific devices that are only available on some of the servers. They cannot be run on the other servers.

At some point, an additional test that combines the switching and the selection is created. From the acceptance point of view, this test need only demonstrate that the switching takes into account selection based on the more complex selection rule. Lots of other combinations of applications and servers may be tested in order to ensure that the code and design do not have more esoteric defects. There would also be acceptance tests for the switching time performance.

Summary

- Separate tests so that each checks a different partsof the flow.
- Separation of concerns makes for simpler testing.

-5

Appendix

This part contains case studies of real-life projects.

Appendix - 1

Money With ATTD

"Money frees you from doing things you dislike. Since I dislike doing nearly everything, money is handy."

Groucho Marx

Developers who have been exposed to TDD often come across the money problem. This problem was introduced in Kent Beck's book, *Test-Driven Development by Example* [Beck01]. An example with unit tests sometimes accompanies many of the xUnit frameworks.

The Context

An organization owns shares of stocks in different companies. The shares are valued in different currencies. You want to add up the money in different currencies and convert it into a total in a given currency. An example of this is as follows, where CHF are Swiss francs.

Total Share Value			
Shares	**Value Per Share**	**Total Value**	
100	1 USD	100 USD	
50	2 CHF	100 CHF	
		150 USD	If 2 CHF = 1 USD

The Original Tests

In Kent Beck's book, he showed how to develop the code using a test-driven approach. The tests he created using JUnit could be expressed in tables. The first table is for currency conversion and demonstrates how Swiss Francs are converted in US dollars.

Currency Conversion		
From Currency	**To Currency**	**Rate**
CHF	USD	2

This table is the setup for the remaining tests. Given this table, we need to be sure that we understand how the rate should be applied. Do you multiply or divide Swiss francs by 2 to get US dollars? So here's an example table for that:

Conversion	
From Amount	**To Amount?**
10 CHF	5 USD

You need to multiply an amount by the number of shares. The tests for this multiplication operation are expressed in this table:

Currency Multiplication		
Amount	**Multiplier**	**Product?**
5 USD	2	10 USD
5 CHF	2	10 CHF

Finally, you need to add two values that may be in different currencies. There are four possibilities. This table expresses two of them. Note that the Sum is always expressed in USD.

Currency Addition		
Amount One	**Amount Two**	**Sum?**
5 USD	5 USD	10 USD
5 USD	10 CHF	10 USD

The first two tests demonstrate our understanding of how addition should work. The other two tests, which were not in Kent's book, may help in further understanding of the addition issue. The sums in the first two cases resulted in USD. Should the sum of two amounts in Swiss francs result in Swiss francs or US dollars? Should the sum always represent USD or should it reflect the currency of the first amount? The sums in this table are shown with "??" to reflect that the answers are uncertain and need further definition.

Currency Addition		
Amount One	**Amount Two**	**Sum?**
5 CHF	5 CHF	10 CHF ??
10 CHF	5 USD	10 USD ??

Now we can express the desired calculation in a table. The Accumulated Value is the sum of the Total Values of the current and preceding rows. [31]

Total Share Value			
Shares	**Value Per Share**	**Total Value**	**Accumulated Value?**
100	1 USD	100 USD	100 USD
50	2 CHF	100 CHF	150 USD

These tables do not drive the details of the underlying implementation. That's what the steps in Test-Driven Design do. These tables were derived from the JUnit tests to demonstrate to the customer how the conversion works. Let's next approach this from the opposite direction.

[31] This table has a second row with values that do not appear in the original tests

The Acceptance Test Approach

Acceptance tests do not indicate how to design a solution. But they are a communication mechanism between the customer, developer, and tester. Let's start by stating the problem as a user story. "As an accountant, I need to convert amounts that are in different currencies to a total in a common currency." The role of accountant is the user with whom we can collaborate in developing acceptance tests.

Our initial collaboration reveals that currencies are converted into other currencies by applying exchange rates. These exchange rates are time dependent. We need a context for the conversion. In the context of accounting, the user says that the exchange rates for this conversion will be fixed at an instance of time. He wants all conversions in the reports he is preparing to use the same exchange rates and that these rates will be those as of midnight on the day prior to the report being prepared.

Decoupling the issue of conversion from the issue of time-dependent exchange rates allows for easier testing. We can create a separate set of tests for the exchange rates.

So we start by making up an exchange rate table. The values in the table are representative of the actual conversion rates. In real life, the conversions may use more digits after the decimal places. They are shown with two decimal digits to keep the example simpler.

Currency To/ Currency From	**EUR**	**USD**	**CHF**
EUR	1.0	1.51	.91
USD	.67	1.0	.73
CHF	1.1	1.35	1.0

Here are examples of how to convert from one currency to another.

Input	**Convert To Currency**	**Converted Amount**
5 EUR	USD	7.55 USD
7.55 USD	EUR	5.06 EUR ??

The accountant came up with the first example, to show what he meant by conversion. The developer created the second. It shows that the reverse conversion produces a different value than the original conversion. This inconsistency causes a discussion. The conversion is not symmetric. Is that what is desired? Should there be separate exchange rates for USD to EUR and EUR to USD? Or should one rate be just the inverse of the other rate? Will the non-symmetry cause any problems? It's time for clarification. This a fundamental issue in the conversion which can affect the rest of the situation. Let's assume that the accountant decided that the conversion should be symmetric. So let's rewrite the setup and the tests.

Currency To/ Currency From	EUR	USD	CHF
EUR	1.0	1.51	.91
USD	X	1.0	.73
CHF	X	X	1.0

Here are the corresponding examples:

Currency Conversion		
Input	**Convert To Currency**	**Converted Amount ?**
5 EUR	USD	7.55 USD
7.55 USD	EUR	5 EUR

The tester suggests a few more examples that come to mind. These examples deal with precision. When converting from USD to EUR, the result turns out to have a large number of digits after the decimal point. The "…" is used to show that the digits do not just stop at "2". What should be done with these digits?

Currency Conversion		
Input	**Convert To Currency**	**Converted Amount?**
5.01 EUR	USD	7.5651 USD ??
7.57 USD	EUR	5.0132… EUR??

These examples bring up a discussion of round-off. How should round-off be handled? Should it be tracked by putting the round-off into a separate account (such as the developer's 401K plan)? Should it be accounted for by some report output? Should the amounts be rounded up or rounded down? Or is the direction of rounding based upon certain conditions? There are certain decisions that can be delayed,

such as what to do with the round-off. The tests for now can at least make sure that it is calculated properly.

Currency Conversion Round Off			
Input	**Convert To Currency**	**Converted Amount ?**	**Round Off?**
5.01 EUR	USD	7.57 USD	.0049 USD
7.57 USD	EUR	5.01 EUR	-.32… EUR

Next to be developed were a few examples of conversion for multiple currencies. The examples looked like:

Currency Conversion With Roundoff			
Input	**Convert To Currency**	**Converted Amount?**	**Round Off?**
5 EUR + 10 USD	USD	17.55 USD	.0 USD
5 EUR + 7.57 USD	EUR	10.01 EUR	-.32… EUR
5.01 EUR + 5.01 EUR	USD	15.14 USD ??	.0098 USD ??

The first two were given by the accountant. The developer wrote the last one. It demonstrates an outstanding issue. Should the amounts be individually converted or should amounts be totaled before conversion? If the former, than the values in the table are correct. If the latter, then the table should be corrected to be:

Input	**Convert To Currency**	**Converted Amount**	**Round Off**
5.01 EUR + 5.01 EUR	USD	15.13 USD	.02 USD

The tester comes up with one more example. This one asks the question of how to handle the round-offs between two different currencies. Should the converted values be added prior to round-off or afterwards? This test does need a second example to show that the underlying implementation does things correctly.

Input	**Convert To Currency**	**Converted Amount**	**Round Off**
5.01 EUR + 5.01 CHF	USD	14.42 USD	.0019… USD

Now the developer has an understanding of the overall picture of the problem. He or she can pick one of the acceptance tests and use that as the starting point for a test-driven design. Unit tests, some of which may be derived from these examples, can check that the classes and methods are giving the desired behavior, such as the round-off. Passing these acceptance tests demonstrates to the accountant that the conversion module as a whole performs as desired.

Appendix 2

Test Framework Examples

"There's more than one way to skin a cat"

Anonymous

There are many acceptance test frameworks. Here are some examples for a few of them for the program in this book.

Test Frameworks

Here are links to more information about acceptance test frameworks. Other test tools, including ones for automating tests at the user interface level, can be found at http://www.opensourcetesting.org/functional.php.

Framework	Website
JBehave	http://jbehave.org/
FIT	http://fit.c2.com/
Fitnesse	http://fitnesse.org/
Easyb	http://www.easyb.org/
Cucumber	http://cukes.info
Robot	http://code.google.com/p/robotframework/
Arbiter	http://arbiter.sourceforge.net/
Concordian	http://www.concordion.org/

The Example

The example tests come from Chapter 10 and 11. They perform the following operations:

- Setup the database with a customer and a CD
- Check-out a CD
- Check-in a CD with a simple rental fee computation.
- Verify the computation of the rental fee for an upcoming change in how rental fees are calculated, based on the CD Category

The Check-out CD and Check-in CD tests are part of a workflow test. The Check-in test assumes that the CD has been checked out. If the tests are run separately or not forced to run in the required sequence, the database setup would be performed for each test. Then the given part of the Check-in test has to put the appropriate values in CD Data.

Fit Implementation

Here is the Fit version. Some tables have been reformatted so they fit on the page. The Fit version with the Java code can be downloaded from: *** Put in a link here ****

***Start background (if it can't be done here because of the header, then do it for each part underneath the header (or maybe no background, since this is almost all a test **)

Database Setup

Setup the database with clean tables

fixtures.SetupDatabase
setup()
true

Add a CD

fixtures.CDs		
PhysicalID	Title	add()
CD1234567890	Beatles Greatest Hits	true

and a customer

fixtures.Customers		
CustomerID	Name	add()
C007	James	true

Check-out CD

Given:

That a CD is not rented

fixtures.CDData				
PhysicalID	Title	Rented	Customer ID	StartTime
CD1234567890	Beatles Greatest Hits	No		

and the date is

fixtures.TestDate	
Date	set()
1/3/10 8:00 AM	true

When:
the CD is Checked Out:

fit.ActionFixture		
start	fixtures.CheckOut	
enter	CustomerID	C007
enter	CDID	CD1234567890
press	Rent	true

Then:
The CD is recorded as rented at the checkout time

fixtures.CDData				
PhysicalID	Title	Rented	Customer ID	StartTime
CD1234567890	Beatles Greatest Hits	Yes	C007	1/3/10 8:00 AM

and the following data is readied to be printed on the Rental Contract

fixtures.RentalContractData					
CustomerID	Customer Name	PhysicalID	Title	RentalDue	check()
C007	James	CD1234567890	Beatles Greatest Hits	1/5/10 8:00 AM	true

Check-in

Given:
at a later date,

fixtures.TestDate	
Date	set()
1/8/10 8:00 AM	true

That the CD is rented

fixtures.RentalContractData					
CustomerID	Customer Name	PhysicalID	Title	RentalDue	check()
C007	James	CD1234567890	Beatles Greatest Hits	1/5/10 8:00 AM	true

When:
the CD is Checked in

fit.ActionFixture		
start	fixtures.CheckIn	
enter	CDID	CD1234567890
press	Return	true

Then:
the CD is recorded as not rented

fixtures.CDData				
PhysicalID	Title	Rented	Customer ID	StartTime
CD1234567890	Beatles Greatest Hits	No		

and charge data is prepared for the charging system. (This is computed at $2 per day)

fixtures.RentalChargeData				
CustomerName	Title	ReturnDate	RentalFee	check()
James	Beatles Greatest Hits	1/8/10 8:00 AM	$10.00	true

Category Based Rental Fees

Given fees based on categories

fixtures.CDCategoryValues				
Category	RentalDays	BaseRentalFee	ExtraDayRentalFee	add()
NewRelease	1	2.00	2.00	true
GoldenOldie	3	1.00	0.50	true
Regular	2	1.50	1.00	true
NotSet	2	1.00	1.00	true

When a rental is returned, then the correct rental charge is computed

fixtures.RentalFee		
Category	Rental Days	Rental Fee()
NewRelease	5	$10.00
GoldenOldie	3	$1.00
Regular	3	$2.50

*** End background

Slim

Here is the Slim version created by Bob Martin. It gives an idea of what text-base tests look like. The text tests are connected to the scenario library. The scenario library in turn talks to methods written for a particular language.
***Start Background** - once again, if it can't be started here, then have it for each section.

Setup

```
-!|CDs|
```

CD ID	Title
CD1234567890	Beatles Greatest Hits

-!|Customers|

Customer ID	Name
C007	James

Check-out CD

```
![ script
Given that CD1234567890 is not rented.
And it is 8:00 on 1/3/2010.
When that CD is rented by C007;
Then it should be marked as rented by C007 at 8:00 on 1/3/2010.
And the rental contract for C007 should have name: James, CD id:CD1234567890, Title:Beatles
Greatest Hits, Due date:1/5/2010, and time:8:00.
]!
```

Check-in CD

```
![ script
Given that CD1234567890 is recorded as rented at 8:00 on 1/8/2010 by C007
When that CD is returned at 8:00 on 1/8/2010.
Then it is recorded as not rented.
And the rental receipt for C007 should have Name: James, Title: Beatles Greatest Hits, Return
Date: 1/8/2010, Return Time: 8:00, Fee: $10.00.
]!
]!
```

Scenario Library

scenario	Given that _ is not rented.	CD
$CD=	echo	@CD
clear rental status of	@CD	

scenario	And it is _ on _.	TIME,DATE
set time	@TIME	
set date	@DATE	

scenario	When that CD is rented by _;	CUSTOMER	
$CCUSTOMER=	echo	@CUSTOMER	
rent cd	$CD	to user	@CUSTOMER

scenario	Then it should be marked as rented by _ at _ on _.	USER, TIME, DATE				
ensure	cd	$CD	is rented to	@USER		
ensure	cd	$CD	was rented at	@TIME	on	@DATE

scenario	And the rental contract for _ should have name: _, CD id:_, Title:_, Due date:_, and time:_.	CUSTOMERID, CUSTOMERNAME, CDID, CDTITLE, DUEDATE, DUETIME										
rental contract for	@CUSTOMERID	should have line with name	@CUSTOMERNAME	cd id	@CDID	title	@CDTITLE	due date	@DUEDATE	due time	@DUETIME	

*** What is Word doing to make these two tables offset ??? ***

scenario	Given that _ is recorded as rented at _ on _ by _.	CD, TIME, DATE, USER				
ensure	cd	@CD	is rented to	@USER		
ensure	cd	@CD	was rented at	@TIME	on	@DATE

scenario	When that CD is returned at _ on _.	RETURN_TIME, RETURN_DATE			
return cd	$CD	at	@RETURN_TIME	on	@RETURN_DATE

scenario	Then it is recorded as not rented.		
ensure	cd	$CD	is not rented.

scenario	And the rental receipt for _ should have Name: _, Title: _, Return Date: _, Return Time: _, Fee: _.	CUSTOMER, NAME, TITLE, RETURN_DATE, RETURN_TIME, FEE	
check	name on receipt for	@CUSTOMER	@NAME
check	title on receipt for	@CUSTOMER	@TITLE
check	return date on receipt for	@CUSTOMER	@RETURN_DATE
check	return time on receipt for	@CUSTOMER	@RETURN_TIME
check	fee on receipt for	@CUSTOMER	@FEE

Category Based Rental Fees

CD Category Values			
Category	rental days	base rental fee	extra day rental fee
NewRelease	1	2.00	2.00
GoldenOldie	3	1.00	0.50
Regular	2	1.5	1.00
NotSet	2	1.00	1.00

Rental Fee		
Category	rental days	rental fee?
NewRelease	5	10.00
GoldenOldie	3	1.00
Regular	3	2.5

**** Add any more that come in. ****

Appendix 3

Calculator Tests

"Arithmetic is where the answer is right and everything is nice and you can look out of the window and see the blue sky - or the answer is wrong and you have to start over and try again and see how it comes out this time"

Carl Sandburg

"You can't become a snowboarder by just reading about it."

Anonymous

To round out the exercises that are suggested throughout the book, here are some exercises you can do. Create acceptance tests for these scenarios. The calculator test exercise gives you an opportunity to create acceptance tests for an existing system and then to use those tests as the basis for creating a new implementation of that system.

Calculator Context

I'm not going to go into much detail about the context. Almost everyone has used a calculator. In some elementary schools, students are required to demonstrate proficiency with a calculator. The type of calculator for which the tests are being written is a simple one with a single memory for a number, such as the one available on the Windows operating system (Figure A3-1)

Figure A3.1 *Calculator*

The table could specify a single input key, with the conditions of the previous step. Note that MS is memory save and MR is memory recall.

Calculator			
Current		**New**	

Memory	Display	Input Key	Display?	Memory?
		2	2	
	2	+	2	
	2	3	3	
	3	1	31	
	31	=	33	
		MS	33	33

Alternatively, the table could state the sequence of steps.

Calculator (Display, Memory Clear)		
Input	**Display?**	**Memory?**
2 + 31 =	33	
2 + 31 = MS	33	33

For the memory save and recall, even more tests could be created.

Calculator (Display, Memory Clear)			
Memory	**Input**	**Display?**	**Memory?**
NA	2 + 3 =	5	
NA	2 + 3 = MS	5	5
5	2 + MR =	7	5

Now you apply a number of different inputs to see what the results might be, such as:

Calculator (Display, Memory Clear)			
Memory	**Input**	**Display?**	**Memory?**
NA	2 + + 3 =	5	
NA	2 + - 3 =	-1	
NA	2 - - 3 =	???	
NA	2 / - 3 =	???	

Since you have an existing application, check the results on that application. Make up a series of examples that match the existing calculator. Now give the examples to a developer and have them create a program that passes these acceptance tests.

Breaking Up Acceptance Tests

If the developer feels that the program is too large to easily complete, you can break up the acceptance tests. One way to do that is to separate tests involving the memory from the other tests. The memory is a separable feature. Another way is to have tests that specify only series of input that are “normal”. That is, they would leave out the sequences that had “/’ followed immediately by “-“. .

Changing Acceptance Tests

The "2 - - 3" test could produce –1 or 5, depending on how the existing calculator works. The developer created a program that passes the test as originally specified. Now change the result so that the other answer is expected. The acceptance test fails. The developer now should change the program to create the expected result.

Do any other tests fail? Do you have redundant tests? Are all tests consistent?

Do you need a user interface? How much effort would it be to run the tests with a user interface than without one?

Other Exercises

Here are some other exercises. Create acceptance tests for them.

Sam's CD Rental

As the Inventory Manager, I want to get a list of all rentals for a CD, so I can see whether it is popular or not.

As Sam, the owner, I want to see a log of all activities such as when things are checked-out and checked-in. This helps me determine whether I need to hire more staff.

As an auditor, I need to be able to keep track of all money transfers for Sam's CD Rental.

As a customer, I want to search for CDs by title, artist, or song.

As a customer, I also want to search for songs whose lengths falls between a minimum and maximum length expressed in seconds.

Triangle

A program inputs three values representing the lengths of the sides of a triangle. The program is to determine whether the triangle is scalene, isosceles, or equilateral.

Zooming

You have a new program that runs on a computer that has finger zooming. You need to check that the zooming works properly and looks okay.

File Copying Exercise

A component is being written for an operating system to copy a file from one directory to another.

Appendix 4

Story Estimating

"What's it worth to you?"

Anonymous

Creating software is about delivering business value. Without some measure of business value, it's hard to determine whether the software has any.

Business Value

Every feature in a system such as Sam's should have business value. Business value can come from numerous areas, such as:

- Increased revenue (sales, royalties, fees)
- Decreased expenses
- Using less resources
- More efficient use of resources
- Customer satisfaction
- Product promoters / satisfiers/ detractors
- Staying in business
- Avoiding risk

Business value in some areas is easy to quantify. It is straight dollars. However in other areas, quantification is more difficult. But without some way to compare these "apples and oranges', it is hard to determine which stories should have higher priority. For example, should a story that saves $10,000 be prioritized over a story that gives some customers more satisfaction?

One way to compare is to assign a relative business value to every story. The customer unit has the responsibility to assign business value. The business value does not have to be an exact measure. It's just relative. If the $10,000 savings seems to be the same as giving customers more satisfaction, then the stories have the same business value.

Relative determination may be made by putting the stories up on the wall one at a time. If a story has more business value than another story, place it above that story. If less value, then put the new story below, or if about the same, on one side. If a story fits between two stories, move the stories and make space for it in-between. For example, in Figure A4.1, Story Two has the highest value, Story One and Story Three are about the same, Story Four less than those, and Story Five seems a lot less.

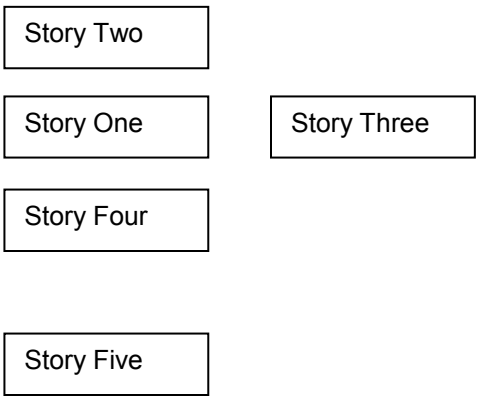


Figure A4.1 *Relative Story Placement*

You can use a Fibonacci series (1, 2, 3, 5, 8, 13 …) or a power-of-two (1,2,4,8,16,32, …) series to assign stories in each row a value. (Figure A4.2). Where you start with the numbering is not important, as long as you assign the same numeric value to equivalent stories in the future.

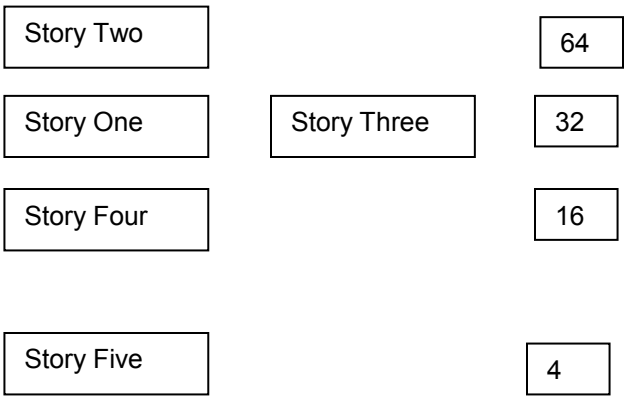


Figure A4.2 *Relative Story Values*

Now periodically, you can measure the cumulative business value of the delivered stories (Figure A4.3) Since the key in agility is to deliver business value quickly, such a chart provides feedback for the developer and tester units to see how well they are doing.

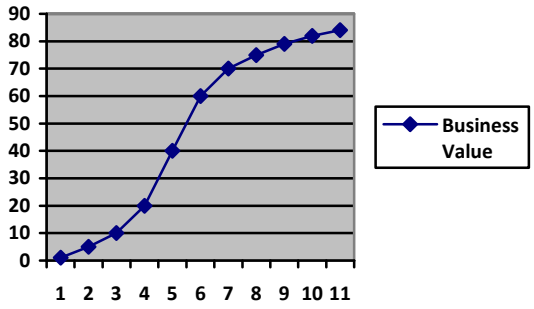


Figure A4.3 *Business Value Chart*

The developer and tester unit can estimate story effort using a similar technique[32]

A story's business value estimate divided by the estimated story effort yields "Bang for the Buck". It is a rough guide to the relative return on investment. It can provide another value when the customer unit makes the decisions as to what stories to develop.

Developer Stories

If a story is estimated to take a long time to implement, it can be broken into smaller stories. However, the customer unit must be able to realize business value from one of the smaller stories. If they cannot, then the smaller stories are developer stories. They exist to make it easier to track units of work. The developers do not get any business value credit until the completed stories deliver something of business value.

As noted in Chapter 16, there should be a developer's created acceptance test for every developer story. Depending on the story, it may not be possible for a specific test, but you should have at least some criteria to be able to know when the story is done.

Is It a Technical Project?

I have seen where a customer story requires a lot of framework development. This translates into a lot of developer stories. If you have this situation, you have an embedded technical project. There should be a "technical product owner" who can break a customer story into "technical customer stories", each with acceptance criteria. And if the framework is vague enough, so these developer stories cannot be created, then you should break this framework out into its own technical project with the customer project being a "test bed" for it, rather than vice-versa.

[32] They could use story poker as described in [Shalloway01].

Summary

- Estimate business value for stories
- Track cumulative business value for delivered stories
- If necessary, break stories into developer stories for tracking
- Business value for a story is not achieved until all developer stories are completed

Appendix 5

Business Capabilities, Rules and Value

"Price is what you pay. Value is what you get."

Warren Buffett

The triad discusses delivering value to the business in other ways.

Business Capabilities

Cathy really would like to provide their customers with discounts for repeat business. She feels that this would keep them loyal to Sam's store. She's heard rumors about a competitive CD rental store that Salvatore Bonpensiero is opening in town the end of the week. Catchy has a vague idea in her mind for new things she can for her customers, maybe giving discounts for renting a lot of CDs.

Debbie notes that the system currently does not keep track of the number of rentals for each customer. She roughly estimates that it would take at least an iteration to start keeping track of that information and then applying that information to the invoices that are sent out.

After hearing Debbie's estimate, Tom asked Cathy if she'd really like it by the end of the week. Cathy replies with a definite yes. Debbie then suggests that Cathy print up some discount cards, as some other retail stores use. The counter clerk can mark off the number of rentals on the card. When the card is filled , the customer would get a free rental.

Cathy loves the idea. It could be in place by the afternoon. She and Tom start discussing ways to avoid a customer misusing the system, such as faking the marks. Debbie suggests putting the Customer ID on the card. Cathy can record in a spreadsheet each day the Customer IDs of the discount cards that have been turned in for a free rental. If a customer has used an abnormal number of cards, Cathy can do some investigation to see if there is an issue that needs to be dealt with. Also Cathy can get a good idea of how many CDs each customer rents to see whether it makes business sense to spend the money on automating the discounts.

Morale of the story: not all capabilities need to be met by software or initially by software. The triad should consider any means that implements the requested capability.

Scenario Handling

Sam is reviewing the demo that Debbie and Tom are presenting. Sam asks a question, "So what happens if something goes wrong with the hardware? Or we have a hurricane which knocks out the power. Or …

". Debbie interjects, "Then we go back to a manual system. You won't have index cards anymore, but we'll have blank contracts that can be filled out by Cary. He'll record the CD ID, the Customer ID, the date rented, the date due, and the amount. We'll keep a printout of all CDs, with their category, the number of days in the base rental period, and the amount. We'll see how long it takes Cary to find information on a CD. If it's really long, you might have a 'power outage' special that has all CDs being rented for the lowest amount and the longest time."

"When the hardware comes back up, Cary or Mary or Larry can enter the information into the system. We'll create a special screen for input of all the information, rather than have the system calculate the return date and the amount. In fact, because of the way we've designed the system for testing, we already have a way to do this in the code. We just have to add a user interface to the program."

Morale of the story: Don't try to deal with every possible scenario in software.

Every Exception Need Not Be Handled

My wife and I travel frequently. When we're on the road, we like to pick up a quick breakfast at a fast food place. We're vegetarians, so that makes it a little more difficult. Usually we stop at McDonald's and order Egg McMuffins, hold the meat. That item usually stops the order taker in his or her tracks. They look like a deer in headlights. They look down at the keyboard. They look back up. There's no key for Egg McMuffin, hold the bacon. Many times they call the manager over to help them. There is some weird combination of keys that allows them to enter such a weird order. It has taken three or four minutes to get the order entered.

One time, we stopped at a Burger King. We order two Breakfast Croissants, hold the meat. The order taker responded almost instantly with the total. I couldn't figure out whether they were better trained, more experienced, or had a better keyboard layout. I looked at the receipt. It said "2 Breakfast Croissants – Ask Cashier". In the event of any off-menu requests, the procedure was to ask the order taker for the details. It sped up taking the order, a great benefit to the customer.

Not handling every exception in software not only saved development time, but also increased customer satisfaction. A win-win.

Business Rules Exposed

A business rule is something that is true regardless of the technology that is employed. Whether rentals are kept on paper or in the computer, Sam wants to limit the number of simultaneous rentals by a customer.

Sam has the business rule that a customer should not rent more than three CDs at any one time. Sam created this rule for a particular purpose. Sam had a limited stock of CDs and he wanted to ensure there was sufficient choice in CDs for other customers. As Sam's inventory grows, the reason for the business rule may change. He may want to increase the limit, so that really good customers who rent numerous CDs are not disappointed. Or he might alter it so that a customer cannot check out two CDs of the same

title. In this case, the business rule might help a customer save money on rentals by avoiding duplicate rentals.

One of the primary purposes of a system is to make transactions comply with the business rules. Business rules should be exposed so that they are easy to test and easy to change.

A Different Business Value

The check-out workflow was modeled from what Sam had noticed in a regular video store. The flow looked like:

***Start background

Customer looks at CD cases
Customer picks the one he wants.
Customer brings it to counter
Clerk retrieves corresponding CD from shelves behind counter and puts case in its place.
Clerk returns to counter with CD.
Clerk scans CD ID and Customer ID
Invoice is printed and signed by customer
Customer heads out door with CD.

***End Background

The triad created a value-stream map from the renter's point of view. The renter wants to rent a particular CD. That is the starting point. The end point is the renter is headed out the door with the CD. The renter wants to get in and out as quickly as possible. So they figured that instead of the customer walking around looking at CD cases, he could just look on a computer screen and select the CD. No more looking for the physical case. So the workflow would be:

***Start Background
Customer selects CD.
Customer enters his Customer ID.
System notifies the clerk that a CD had been requested by the customer and prints an invoice.
Clerk gets the CD and places it in a check-out box with the invoice.
Customer walks up with Customer card.
Clerk retrieves CD and invoice and gives it to the customer.
Invoice signed by customer
Customer heads out door with CD.
***End Background

This workflow is quicker and faster for both the clerk and the customer. Sam also figured he could save a large chunk of store rental by eliminating the shelves with all the CDs. The clerks wouldn't have to replace the CD cases when the CD was returned. By looking at a larger value stream, the triad found a win-win situation for everyone. Software teams that look at the bigger workflow may find non-software improvements.

Summary

- Not all capabilities need to be met by software solutions.
- Software does not need to be able to handle every scenario.

- Software teams should examine workflows in which their software participates for bigger picture improvements.

Appendix - 6

This material has been moved to Appendix 3.

Appendix - 07

Tables Everywhere

"It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

Abraham Maslow

"Everywhere around the world. There'll be dancing in the streets"

The Mamas and The Papas

Tables can be used to drive manual acceptance tests and to clarify not just functional requirements, but "illity" requirements as well.

User Interface Tests With Tables

In Chapter 4 Debbie and Tom presented the test cases used for a discount business rule. The cases were:

Discount		
Item Total	**Customer Rating**	**Discount percentage?**
10.00	Good	0
10.01	Good	1
50.01	Good	1
.01	Excellent	1
50.00	Excellent	1
50.01	Excellent	5

That chapter presented several ways to execute the test. One way was to use a test script for the user interface. No test script was presented in that chapter. You could express the test script with tables that give the information needed to run the test. For example, the following tables have the setup necessary for all the test cases, and an example of the table flow for the first test case.

***Start Background
Given these items

Items For Sale	
Item Number	**Cost**
I1	10.00
I2	10.01
I3	50.01

I4	.01
I5	50.00
I6	50.01

and these customers

Customers		
Name	**Rating**	**Password**
George	Good	12345678
Edward	Excellent	11111111

When a customer logs on

Logon		
Enter	**Name**	George
Enter	**Password**	12345678
Press	**Submit**	

And adds an item to the shopping cart

Add Item		
Enter	**Item Number**	I1
Press	**Add**	

And checks out

Checkout	
Press	**Checkout**

Then the discount on the order should match the percentage in the Discount table.

Order			
Item Total	**Discount?**	**Discount Percentage? (Discount / Item Total)**	
$10.00	$0.0	0 %	

***End Background

The setup for this test could be part of a standard setup for all the tests for the system. It would usually not go through the user interface, but be a programmatic setup as shown in Chapter 20.

This test could be executed manually or automatically. When driven manually, the tester should be able to determine from the screens how to enter the information. If not, then the user interface probably has issues.

As an automatic test, each table could be associated with a script that drives the corresponding user interface screen. These scripts could then be reused in other tests that employ the same tables with the same or different data.

Requirement Tables

The original requirement for the discount was expressed in textual form. The requirement itself could be put into a table for clarification. [Parnas01]. Here was the original discount business rule:

*** Production Note – the indentation on this paragraph is non-standard.
If Customer Type is Good and the Item Total is less than or equal $10.00,
Then do not give a discount,
Otherwise give a 1% discount.
If Customer Type is Excellent,
Then give a discount of 1% for any order.
If the Item Total is greater than $50.00,
Then give a discount of 5%.

In table format, this rule could look like one of the following tables. The form of the table should be whatever is appropriate to the problem, as discussed in Chapter 21.

Discount Rule		
Type	**Item Total**	**Discount Percentage**
Good	<= $10.00	0.0%
Good	Otherwise	1.0%
Excellent	Any	1.0%
Excellent	> $50.00	5.0%

Or it might be like:

Discount Rule		
Type	**Item Total**	**Discount Percentage**
Good	<= $10.00	0.0%
	Otherwise	1.0%
Excellent	Any	1.0%
	> $50.00	5.0%

The table format clarifies what was potentially unclear or ambiguous in the text. For example, the "any" of the original text statement appears more unclear when it is placed in this table. In addition, filling out the tables may bring to light suppressed premises or unstated requirements or assumptions. In this example, having a column for type may suggest that there are other types for which the discount rule may be applicable.

Another Table

To be complete, the discount percentage example assumes that a customer has a rating of Good or Excellent without specifying how that rating was arrived at. The separation of concerns of what discount to apply from the determination of the customer type makes the tests for both more robust.

Similar to what was done in Chapter 13, here is a business rule for determining the Customer Rating.

Customer Rating Rule	
Total of Orders	**Customer Rating**
<= $1000	Regular
>$1000	Good
> $5000	Excellent

When rules are expressed as tables, making up the tests often is straightforward. In this example, you simply create tests at each of the transition points between each rating.

Customer	
Total of Orders	**Customer Rating?**
$1000	Regular
$1000.01	Good
$5000	Good
$5000.01	Excellent

Now you do need a test that shows that the Total of Orders for a particular customer is correct. Making up tables showing a set of orders for customers that total up to the amounts in this table is left as an exercise for the reader.

Illity Requirements

You can use tables to indicate the required "illity" measures that an application must meet. For example, in Chapter 12, Tom showed one for performance of check-outs.

Check-out Performance	
Number of simultaneous check outs	**Response Time Maximum (seconds)**
1	.1
10	.2
100	.3

If platform capacity was a constraint, then the triad could make up a resource table, such as:

Maximum Resources		
Number of users	**Memory in Megabytes**	**CPU Cycles in Million of Instructions Per Second**
1	10	1
10	30	5
100	50	20

Data Tables

Can you use the customer tables as the data? Sure. You need to have a translation mechanism, but that's fairly simple. For example, in Chapter 10, there was a table for rental rates:

Rental Rates			
CD Category	**Rental Period Days**	**Rental**	**Extra Day**

		Rate	Rate
Regular	2	$2	$1
GoldenOldie	3	$1	$.50
HotStuff	1	$4	$6

This table could be the actual source for the rates. The program would read this table and use the values as the rates. If Sam wants to change the rates, he just changes this table and tells the program to re-read it.

Summary

- You can use tables to drive user interface tests manually or automatically
- Use tables for requirements for clarification and exposing unexpressed details

50

Epilogue

"All good things must come to an end"

English Proverb

Acceptance test driven development has been presented through stories and examples. Here is the final word.

Who, What, When, Where, Why, and How

We stated in the preface the context of the book and the questions it answers. Here are the questions again and if you skipped the entire book, here are the answers.

- What – Acceptance criteria for projects and features, acceptance tests for stories
- Who – The triad – customer, developer, and tester communicating and collaborating
- When – Prior to implementation – either in the iteration before or up to one second before, depending on your environment
- Where – Created in a joint meeting, run as part of the build process
- How – In face-to-face discussions, using Given/When/Then and tables
- Why – To effectively build high quality software and to cut down the amount of rework

So What Else Is There?

Feel free to write to me at ken.pugh@netobjectives.com if you have comments or questions. Let me know what topics you would like covered in more depth. With the web world, it's easier to increase the amount of information. If your triad needs an in-person run-through of creating acceptance tests, Net Objectives offers courses and coaching. (www.netobjectives.com)

Try It Yourself

Based on Sam's growing business, some readers may decide they want to open a CD rental store. For information regarding the legality of renting CDs in the U.S., please see 17 USC 109 (b)(1)(A), In Japan, it is legal to have a licensed CD rental store that pays royalties to JASRAC (Japanese Society for Rights of Authors Composers and Publishers).[Isolum01]

> Acceptance test driven development:
> Be sure you have the right answer before implementation.
> The answer is not just 42.
> It's 4 8 15 16 23 42.

Thoughts Of Others

Chris Tavares says “The advantage of using acceptance tests is that, in order to properly code the tests, it requires the customer to actually nail down what the requirement is. It also gives an unambiguous indication that the requirement is actually fulfilled. It may not be sufficient, but it's a heck of a lot better than just a functional spec.” [Tavares01]

Danny Faught refers to a contract. Acceptance tests are a specific form of a contract. “Contract reflects developers understanding of the business domain in a more explicit way, so it makes it easier to spot misjudgments developers might have done in their implementation. This intention revealing method of development highlights important business rules, rather than letting them be hidden behind an abstraction. Abstractions should not hide any important business concern as an implementation details. Explicit contracts and invariants allow business experts to have a look on what is going on inside the software.” [Faught01]

51

References

"And now for the rest of the story"

Paul Harvey

Here are books and websites that were referenced in the book, as well as other good books on testing and the software development process.

Referenced

[Wiki01] http://en.wikipedia.org/wiki/N-body_problem

[Wiki02] http://en.wikipedia.org/wiki/Four_color_theorem

[Answers01] http://www.answers.com/topic/acceptance-test

[Crispin01] http://lisacrispin.com/wordpress/tag/power-of-three/

[Sutherland01] http://jeffsutherland.com/JakobsenScrumCMMIGoingfromGoodtoGreatAgile2009.pdf

[Poppendieck01] http://www.poppendieck.com/design.htm

[Weinberg01] Exploring Requirements: Quality Before Design; Donald C. Gause, Gerald M. Weinberg

[Chelimsky01] The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends by David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North (The Pragmatic Bookshelf)

[Marick01] http://www.exampler.com/

[Kerievsky01] http://industriallogic.com/papers/storytest.pdf

[Evans01] Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans

[Beck01] Beck, Kent, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[Parnas01] http://www.cs.iastate.edu/~colloq/new/David_Parnas_slides.pdf

[Poppendieck02] Poppendieck, Mary and Tom Poppendieck Implementing Lean Software Development: From Concept to Cash, Addison-Wesley Professional, 2006.

[Poppendieck03]http://www.poppendieck.com/papers/LeanThinking.pdf

[Poppendieck04] http://www.poppendieck.com/pdfs/Lean_Software_Development.pdf

. [Agile01] http://agilemanifesto.org/principles.html

[Cockburn01] Cockburn, Alistair, Agile Software Development: The Cooperative Game, Addison Wesley Professional 2006

[Meszaros01] Meszaros, Gerard, xUnit Test Patterns: Refactoring Test Code, Addison-Wesley, 2007.

[Wiki03] http://en.wikipedia.org/wiki/Software_architect

[Answers02] http://www.answers.com/topic/system-test

[Jefferies01] http://xprogramming.com/articles/expcardconversationconfirmation/

[Koskela01] http://www.methodsandtools.com/archive/archive.php?id=72

[Coplien01] Bjørnvig, Gertrud, James Coplien and Neil Harrison. "A Story about User Stories and Test-Driven Development". Better Software 9(11), November 2007, ff. 34. http://www.rbcs-us.com/images/documents/User-Stories-and-Test-Driven-Development.pdf

[Fast01] http://www.fastcompany.com/blog/charles-fishman/usair-asks-fliers-%E2%80%98can-we-get-hallelujah%E2%80%99

[Pettichord01] Kaner, Clem and James Bach and Bret Pettichord, Lessons Learned in Software Testing, Wiley, 2001.

[Constatine01] Constantine, Larry, and Lucy A.D. Lockwood, Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design, Addison-Wesley Professional, 1999.

[Wiki04] http://en.wikipedia.org/wiki/Smoke_test

[Cunningham01] R Mugridge & W Cunningham, *Fit for Developing Software: Framework for Integrated Tests,* Prentice Hall PTR (2005).

[Cunningham02] http://fit.c2.com/wiki.cgi?FrameworkHistory

[Martin01] http://fitnesse.org

[Usability01] http://www.usabilitynet.org/trump/documents/Suschapt.doc.

[Craig01] http://www.mockobjects.com/files/endotesting.pdf, EndoTesting: Unit Testing with Mock Objects by Tim Mackinnon, Steve Freeman, and Philip Craig

[Hussman01] David Hussman, Practical Agility. The Pragmatic Bookshelf (in production 2010)

[Patton01] http://www.agileproductdesign.com/presentations/user_story_mapping/index.html

[ABA01] http://www.abajournal.com/news/article/judge_calls_for_end_to_lawyers_obfuscation_suits_madness/

[Security01] fhttps://www.pcisecuritystandards.org/security_standards/pci_dss.shtml

[IBM01] http://publib.boulder.ibm.com/infocenter/rbhelp/v6r3/index.jsp?topic=/com.ibm.redbrick.doc6.3/wag/wag29.htm

[Cunningham03] http://c2.com/wikisym2007/pnsqc2007.pdf

[Gottesdiener01] Gottesdiener, Ellen, Requirements by Collaboration: Workshops for Defining Needs, Addison-Wesley Professional, 2002.

[Gottesdiener02] http://ebgconsulting.com/articles.php#wkshp

[Gottesdiener03] http://ebgconsulting.com/facassets.php

[Cockburn02] Cockburn, Alistair, Writing Effective Use Cases, Addison Wesley Professional, 2000

[Mindtools01] http://www.mindtools.com/CommSkll/ActiveListening.htm

[Wiki05] http://en.wikipedia.org/wiki/Myers-Briggs_Type_Indicator

[Project01] http://www.projectsmart.co.uk/smart-goals.html

[Martin02] http://butunclebob.com/FitNesse.UserGuide.FitLibraryUserGuide.DoFixture

[Riordan01] Heads First Ajax Rebecca M. Riordan

[Pugh01] Ken Pugh, Interface-Oriented Design

[Wiki06] http://en.wikipedia.org/wiki/Unified_Modeling_Language

[Ambler01] Ambler, Scott, W., Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process, Wiley, 2002.

[Fowler01] Fowler, Martin, UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition) Addison-Wesley Professional;2003.

[Feathers01] Feathers, Michael, Working Effectively with Legacy Code, Prentice Hall, 2004.

[Hunt01] Hunt, Andrew, and Dave Thomas, The Pragmatic Programmer: From Journeyman to Master , Addison-Wesley Professional 1999.

[Pugh02] Ken Pugh, Prefactoring - Extreme Abstraction, Extreme Separation, Extreme Readability.

[Systems01] http://www.systems-thinking.org/rca/rootca.htm

[Wiki07] http://en.wikipedia.org/wiki/Root_cause_analysis

[Cohn01] Cohn, Mike, Agile Estimating and Planning, Prentice Hall, 2005.

[Eckstein01] Agile Software Development with Distributed Teams: Staying Agile in a Global World by Jutta Eckstein

[Tavares01] http://www.tavaresstudios.com

[Faught01] http://sadekdrobi.com/

[Satir01] http://www.satirworkshops.com/files/satirchangemodel.pdf

[IEFF01] http://www.ietf.org/rfc/rfc2822.txt

[Security01] http://www.securiteam.com/securityreviews/5DP0N1P76E.html

[Wiki08] http://en.wikipedia.org/wiki/Sound_level_meter

[OSHA01] http://www.osha.gov/dts/osta/otm/noise/health_effects/physics.html

[Constatine02] W. Stevens, G. Myers, L. Constantine, "Structured Design", IBM Systems Journal, 13 (2), 115-139, 1974.

[Shalloway01] Achieving Enterprise Agility Alan Shalloway, James R. Trott, and Guy Beaver

[Isolum01] http://lsolum.typepad.com/copyfutures/2004/10/a_lesson_from_j.html

** Need to find where to reference these

[Fowler02] Fowler, Martin, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002. - enterprise

[Rising01].Manns, Mary Lynn and Linda Rising, Fearless Change: Patterns for Introducing New Ideas, Addison-Wesley Professional, 2008. =*** Add to in Chapter 26 !!

[Crispin02] Crispin, Lisa and Janet Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley Professional, 2009. -***Add to in an early chapter

[Beust01] Next Generation Java Testing: TestNG and Advanced Concepts

CÃ©dric Beust and Hani Suleiman - add to Chapter 4

Other Good Books About Testing and the Software Development Process

[Osherove01] The Art Of Unit Testing By Roy Osherove

[Astels01] Test-Driven Development, A Practical Guide by David Astels

[Williams01] An Initial Investigation of Test Driven Development in Industry by Bobby George and Laurie Williams

[Adzic01] Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing by Gojko Adzic

[Stott01] Using FIT inside Visual Studio Team System

Visual Studio Team System: Better Software Development for Agile Teams

By Will Stott, James W. Newkirk

[Elssamadisy01] Chapter 44 in Agile Adoption Patterns: A Roadmap to Organizational Success by Amr Elssamadisy Chapter 44 – Test-Driven Requirements

[Wiegers01] Karl Wiegers, Software Requirements,, 2nd Edition Microsoft Press, 2003.

[Wiegers02] Karl Wiegers, More About Software Requirements: Thorny Issues and Practical Advice, Microsoft Press, 2006

[Coplien03] Bjørnvig, Gertrud, James Coplien and Neil Harrison. "A Story about User Stories and Test-Driven Development, Chapter 2: Into the Field". Better Software 9(12), December 2007, ff. 32.

[Janzen01] Janzen and Saledian, "Does Test-Driven Development Really Improve Software Design Quality?" IEEE Software 25(2), March/April 2008, pp. 77 - 84.

[Siniaalto01] Siniaalto and Abrahamsson, Comparative Case Study on the Effect of Test-Driven Development on Program Design and Test Coverage, ESEM 2007.
[Siniaalto01] Siniaalto and Abrahamsson. Does Test-Driven Development Improve the Program Code? Alarming results from a Comparative Case Study. Proceedings of Cee-Set 2007, 10 - 12 October, 2007, Poznan, Poland.

[Shore01] Shore, James and Shane Warden, The Art of Agile Development , O'Reilly Media, 2007.

[Kerth01] Kerth, Norman L., Project Retrospectives: A Handbook for Team Reviews,Dorset House Publishing Company 2001.
[Richardson01] Richardson, Jared and William A. Gwaltney, Ship it! A Practical Guide to Successful Software Projects Pragmatic Bookshelf; 2005.

Wake, William C. Extreme Programming Explored, Addison-Wesley Professional 2001.

[Demarco01] Demarco, Tom Demarco, Peter Hruschka, Tim Lister, and Suzanne Robertson Adrenline Junkies and Template Zombies: Understanding Patterns of Project Behavior, Dorset House, 2008

[Rainsberger01] Rainsberger, J.B., JUnit Recipes: Practical Methods for Programmer Testing, Manning Publications, 2004.

[Grenning01] James W. Grenning, Test Driven Development for Embedded C

[Copeland01] Lee Copeland, A Practitioner's Guide to Software Test Design

[Weinber02] Perfect Software: And Other Illusions About Testing by Gerald M. Weinberg

[Galen01] Software Endgames: Eliminating Defects, Controlling Change, and the Countdown to On-Time Delivery by Robert Galen

[Pardee01] To Satisfy & Delight Your Customer: How to Manage for Customer Value by William J. Pardee

[Rothman01] Hiring the Best Knowledge Workers, Techies & Nerds: The Secrets & Science of Hiring Technical People by Johanna Rothman

[DeMarco02] Waltzing with Bears: Managing Risk on Software Projects by Tom DeMarco and Timothy Lister

[Marick02] Marick, Brian, The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing, Prentice Hall, 1994.

[Highsmith01] Highsmith, Jim, Agile Software Development Ecosystems, Addison-Wesley Professional, 2002.

[Jefferies01] Jeffries, Ron. Extreme Programming Installed, Addison-Wesley Professional, 2000.

[Poppendieck02] Poppendieck, Mary and Tom Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003.

[Poppendieck03] Poppendieck, Mary and Tom Poppendieck, Leading Lean Software Development: Results Are not the Point , Addison-Wesley Professional, 2009.

[Coplein05] Coplein, James O. and Neil B. Harrison, Organizational Patterns of Agile Software Development, Prentice Hall, 2004

[Ambler01] Ambler, Scott, W., Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process, Wiley, 2002.

[Wake01] Wake, William C. Extreme Programming Explored, Addison-Wesley Professional 2001.

[Gause02] Gause, Donald C and Gerald M. Weinberg, Exploring Requirements: Quality Before Design Dorset House Publishing Company, 1989

[Williams01] Williams, Laurie, and Robert Kessler, Pair Programming Illuminated, Addison-Wesley Professional, 2002.

[Kerievsky01] Joshua Kerievsky, Refactoring to Patterns

[Bain01] Emergent Design: The Evolutionary Nature of Professional Software Development by Scott L. Bain (2008)