Using Customer Tests to Drive Development

Articles

Posted by:
Posted on : 2007/9/7 2:56:36

Like many agile software development teams, our team writes tests for each feature before the feature is actually developed. We’ve found many advantages to using tests to drive development, not only at the unit test level but at the functional, system and acceptance test levels. Not only do we have tests which show whether we’ve delivered the correct functionality, but we benefit from increased communication and collaboration, increasing the chances that we will deliver exactly what our customers want. Writing just the right amount of tests and level of detail has proved difficult at times, as has the automation and timing of the automation effort. The effort to overcome those problems has paid off and led us to devote even more resources to driving development with customer tests.

Author: Lisa Crispin, http://lisa.crispin.home.att.net

Test-driven development or TDD is a widely accepted practice used by agile software development teams of many flavors – not only Extreme Programming teams. For each small bit of functionality they code, programmers first write unit tests, then they write the code that makes those unit tests pass. TDD is seen as a design tool, since it forces the programmer to think about many aspects of each feature before coding. It results in a ‘safety net’ of tests that can be run with each build, ensuring that new code doesn’t ‘break’ any existing code, or that refactored code maintains its functionality.

Why Customer Test-Driven Development?

I’ve been fortunate to work as a tester on three development teams practicing TDD as they produce J2EE-based web applications. Code produced via TDD far outshines code produced in a ‘traditional’ manner in quality and robustness. Testing code developed in this manner, I don’t find the bugs I was used to finding in ‘typical’ projects, such as bugs produced by boundary conditions or ‘invalid’ input. Don’t get me wrong, as a tester, this makes me happy. So what more do I want?

What I still saw, even with TDD, are misunderstandings between the project’s customers (also known as business experts or product owners) and the software developers. Even if the deployed code was almost bug-free, it didn’t do everything the customer had expected. We can enlarge on the concept of TDD and reduce the risk of delivering the ‘wrong’ code with what I call customer test-driven development.

Customer test-driven development (CTDD), also known as story test-driven development (SDD), consists of driving projects with tests and examples that illustrate the requirements and business rules. What do I mean when I say ‘customer tests’? Basically, anything beyond unit and integration (or contract) testing, which are done by and for the programmers, testing small units of code and their interaction. I use the term ‘customer’ in the XP sense, meaning

product owners and people on the business side who specify features to be delivered. Customer tests may include functional, system, end-to-end, performance, load, stress, security, and usability testing, among others. These tests show the customers whether the delivered code meets their expectations.

How CTDD Works

How do we know everything that the customer expects ahead of time? On agile projects, we make this easier by splitting features into small, manageable chunks, known as ‘stories’. Often written on a small index card, stories take a form such as this:
* As a retail website customer, I would like to be able to delete items out of my shopping cart so that I can check out with only the items I want.

For each story, we ask the customer to tell us how she will know when this story is complete, in the form of tests and examples. Just as with TDD, before writing any code, we write tests that, when passing, prove the code meets the minimum requirements. These tests are ideally in a form that can be used with an automated tool, but they may also be higher-level tests, or guidelines for later exploratory testing.

Who are these cooperative customers? They’re the people with the domain expertise, who understand business priorities. They may be in sales or marketing, they may be business managers or analysts, they may be end users, they may even be a development manager or customer support. They will most likely need help in writing effective tests that the programmers can use.

Testers provide this help, combining their understanding of the technical requirements of the system with the big picture of what the business needs. Testers ask questions to help elicit requirements and flush out hidden assumptions. For example, with the story above to delete items out of the shopping cart, testers may ask:
* Should there be a dialog for the user to confirm the delete?
* Is there a need to save the deleted items somewhere for later retrieval?
* Can you draw a picture of how the delete interface should look?
* What should happen if all items in the cart are deleted?
* What if the user has two browser sessions open on the same cart, and deletes an item in one of the sessions?

By writing customer tests ahead of coding the features, we can bring hidden assumptions to the surface. Frequent conversations with our customers, going over examples, enable our product owners to get the best possible results.

Teams doing TDD, especially those trying it for the first time, may tend to do only the more obvious, "happy path" tests. Misunderstood requirements and hard-to-find defects may go undetected. Writing customer tests first provides navigation for our project ‘road trip’. The tests help the team identify milestones and landmarks to know if they’re on track. When the tests all pass, we know we’ve reached our destination.

CTDD in Real Life

The basic process of CTDD sounds simple. Once stories are identified for an iteration, we can specify customer tests, write the fixtures which automate the tests, then write the code that makes the tests pass.

First, you need some way to specify the tests and then automate them. Our team uses a tool called FitNesse. FitNesse is an open source software development collaboration tool which enables customers, testers and programmers to specify test cases as simple tables of inputs and expected outputs. It is a wiki, which means it is easy to create and edit pages in a web browser without having to know HTML. Programmers find it easy to write test fixtures which operate the code and return the results, using the same language as the application under test (for example, Java).

Even with an appropriate lightweight tool such as FitNesse, specifying and then automating tests can be a huge challenge, as my team found when we decided to try CTDD. First of all, no matter how easy it is to specify tests in the tool, it’s hard to make time to write tests in advance of, or even at the very beginning of an iteration. We were already struggling with TDD, and felt short of resources. When we faced a big upcoming theme to develop, our customer and I made writing tests in advance a priority. The customer wrote detailed examples in spreadsheet form, and I turned these into FitNesse test tables. Not surprisingly, we made some big mistakes.

When the programmers started writing the automated fixtures for the first story, they found the large number of highly detailed level of the tests overwhelming. It was hard for them to get a big picture of what how the highly complex business rules should work. Also, once they started writing the fixtures to automate the tests, the programmers needed a major redesign to the test cases. Since I had written so many tests in advance, it was a big and tedious job to go back and refactor all the tests. We also had disagreement in where business logic should be located. The programmers originally wanted to put much of the logic into the SQL that retrieved records for processing, but this made test automation more difficult. This turned into a bit of a crisis, and we wondered if CTDD was worth the investment. In spite of this, the FitNesse tests which finally emerged proved to be valuable.

In hindsight, even this small crisis and ensuing discussion proved a valuable learning experience. We finally agreed that putting the complex business logic into the Java code was the safest course of action. It could be tested more easily and thoroughly. We also agreed that, at least for our project, writing detailed tests well in advance of coding didn’t serve the purpose of helping the programmers know what to code. It also led to wasted time refactoring tests later.

We decided to write only high level tests in advance of each development iteration, or during the first couple days of each iteration (we use two week iterations). These test cases, written by the business experts and/or testers, usually consist of bullet points, narrative and/or examples on a wiki page. Often, our business experts will explain the story to the team and write examples on a whiteboard. Customer tests don’t replace direct communication between programmers and customers, they enhance it and document the story requirements. The programmers use these high-level tests to understand each story and help with the design. The high-level test wiki page might look something like this:

Story: As a retail website customer, I would like to be able to delete items out of my shopping cart so that I can check out with only the items I want.

Notes:

A delete box is displayed next to each item in the shopcart. An ‘update cart’ button is at the bottom of the list. If the user marks the delete box for one or more items and clicks the ‘update cart’ button, a window pops up giving the user the option to delete or move the item(s) to his wishlist. This popup window also has a cancel button.

Test cases:
* Click the update cart button without checking any items. Should refresh the page.
* Delete one item. Verify it is gone from the cart and not in the wishlist.
* Move one item to the wishlist. Verify it is there and no longer displayed in the cart.
* Delete all the items. The "keep shopping" button should appear.
* (and so on – you get the idea).

As the iteration gets underway, we write FitNesse tests for each story. The other tester on the team and I do this, collaborating closely with our business experts, who often prepare examples and test cases in spreadsheets. We use the build-operate-check pattern for our tests. Each test page starts off with tables that build inputs to the story. These inputs may be global parameters, data stored in memory to be used by the test, or if necessary, actual data in the database. Then we invoke a method (not written yet) which will operate on the inputs with the actual code. Last, the test contains tables which verify the results, again either reading data from memory or from the database itself. Each test also has a setup and teardown method.

For the sample story above, about deleting items from a shopcart, the test might be organized this way:
1. Build a shopcart into memory, adding all the fields for the item such as item number, description, price, quantity.
2. Operate on the shopcart, specifying one item to delete.
3. Check to see that the shopcart contains the correct remaining items.

Unless we’re fairly certain of the test design, we only write one or two test cases, so that we don’t lose a lot of work if we need a big refactoring later. We show the test to a programmer and discuss whether it’s a good approach, making changes if needed. In true CTDD, the programmers would write the test fixture to automate these tests before writing the production code, the same as they do with unit tests. In our project, the programmers usually look over the high-level test cases and write their first draft of the code. Then they take on the task to automate the FitNesse test. Often, the test case shows a requirement that they neglected or misunderstood, and they go back and change the code accordingly. Once the methods for the FitNesse tests are working, we can go back and add test cases, often uncovering more design flaws or just plain bugs.

Once our FitNesse tests are passing, they become part of our daily regression test suite, catching any flaws introduced later. But their most important function has already been served: Writing the tests has forced communication between customers and testers, customers and programmers, testers and programmers. The team had a good understanding of the story’s requirements before starting to write any code, and the resulting code has a good chance of meeting all the customer’s expectations.

Building on CTDD

Despite the success whenever we write customer tests first, we can become complacent or get in too much of a rush and neglect this practice. Recently we had a fairly simple story (we thought) that only a couple of internal customers would use. We did write test cases in advance of coding, but as the internal customers were always busy and we thought we understood it well, we neglected to go over the test cases with the customers. After deploying to production, one of the customers tried to use the new functionality, and found it was not at all what he wanted. Now we have an unhappy customer, who has to write new stories to get what he wanted in the first place.

When our business was at its peak time of year and we were all busy, sometimes the tasks to automate tests got pushed towards the end of the sprint, making a time crunch for testing. We made ‘writing FitNesse fixtures early’ a team goal for a few iterations, until this became a habit. We use a task board to monitor progress for each story, and if the ‘write FitNesse fixtures’ task doesn’t move into the Work In Progress column by a certain point, someone will ask about it during the standup meeting.

Our company has experienced the benefits of CTDD, and has made its use a company-wide goal. Although our company is quite small (only around 20 employees), we have a product owner (who is also our ScrumMaster) who works with our team full time. In addition, a senior vice-president who is the most knowledgeable about the domain has been given the time and directive to work closely with our engineering team. He writes test cases for at least one story in advance of each iteration. We have a new goal to finish high-level test cases for all stories of an iteration by the fourth day of the two-week iteration. This new commitment to CTDD, along with our commitment to early test automation, makes us talk to each other when we need to – before and during coding.

Other Success Stories

Our team has drawn inspiration from other teams’ use of CTDD. Richard Watt and David Leigh-Fellows presented their techniques in "Acceptance Test Driven Planning" at XP/Agile Universe 2004. They use a practice they call "Getting our stories straight". QA works with customers at end of each iteration to write tests for next. They use these tests as a planning tool to guide estimation and task breakdown. It’s not Big Up-Front Design (BUFD), but a "just right" amount of process.

Another source of ideas was an article in Better Software Magazine (July/August 2004) by Tracy Reppert called "Don’t Just Break Software, Make Software". She details the story test-driven development used by Nielsen Media Research. Joshua Kerievsky, a pioneer of STDD, says it "helps teams obtain the necessary amount of story details before writing code." He also notes, "Following story test driven development clearly enabled me to produce designs I simply would not have anticipated."

What if I’m not On an Agile Team?

"It all sounds great", you say, "but I’m stuck in a traditional waterfall process. How can CTDD benefit my project?" After being on XP teams for a couple years, I had to go back to the "dark side" and work on a less agile team for a time. They wanted to implement agile development, but couldn’t quite make the commitment. I asked the managers where they felt the most pain on projects. They immediately responded that requirements were their biggest problem. On some projects, too much time was spent gathering requirements, which were changed and out of date right

away and thus useless. On the rest, the programmers were forced to start coding with no requirements at all.

"What if we write customer tests ahead of development?" I proposed. They agreed to try it, and our results were good. We were able to get people on the business side working more closely with us, and the programmers appreciated having some direction before they started writing code. Even though the programmers didn’t assist directly with test automation (all automation was done by my test team), writing customer tests first did help guide development. The resulting software was closer to the customer’s requirements. This success led to trying more agile practices on ensuing projects, again with good results. Writing tests ahead of development is something a test team can implement without a major effort, it just requires a bit of cooperation from the business side. It can change a team’s culture just a bit, so that it may be open to more practices that will improve development, and improve life for the developers! Originally published in the Summer 2005 issue of Methods & Tools