A Story About User Stories and Test-Driven Development

Chapter 2: Into the Field
Stories driven Development

by Gertrud Bjørnvig, James O. Coplien, and Neil Harrison
In the November 2007 issue of Better Software magazine, we introduced a small team that was using test-driven development (TDD) as its design method. The team is a fictional amalgam of our consulting clients, but typifies the experience of our clients and the experiences related by our colleagues who are project managers, ScrumMasters, and developers. We found that our metaphorical project struggled to deliver a product and that many of the struggles were attributed to TDD itself. These are not small problems and are not limited to testing or even to design. A TDD-driven approach traditionally builds on user stories instead of use cases, which leads to poorly specified systems and an inordinate amount of rework. It leads to very spotty coverage of the design space and ignores the most important aspect of design: the interactions between the parts. It causes developers to become blindsided—developing code and tests guaranteed to validate each other.

Some of these problems owe to common practice, which TDD founders may discount as misguided. However, that makes these practices no less common, whether because the development world can’t understand the subtleties beneath TDD or because all the consultants and lecturers get it wrong. We have attended many tutorials on TDD during the past year, and our story reflects practices that they advise. But some of the problems telegraph a short-sightedness in TDD itself: the need for short-term gratification—er, feedback—that too often ignores serious, long-term consequences.

Our team struggled to get its software to the field. We congratulate them and pick up the story where we left off in Chapter 1.
Another customer comes along with a killer bug. The development team goes in and tries to find it, but the system is doing exactly what it is supposed to be doing; all the tests pass, but the system doesn’t work right. After three days of head scratching, the team finds that one of the tests is wrong—just a coding error in the Java code for the test. Oops. Once the test is fixed, it fails. It turns out that this is a fairly common case, and according to Thomas Christensen, the word on the street is that at many TDD companies the motto is: “We don’t fix bugs, we fix tests.” This brings us to the first risk of Chapter II:

According to Matt Stephens and Doug Rosenberg, in a typical TDD setting, half the code mass comprises unit tests. Those tests are code. Where there is code, there are bugs (see the StickyNotes for a reference). Programmers are psychologically programmed to trust their tests over their code—it takes only five minutes to write the test and an order of magnitude longer to write the code. They sometimes will make arbitrary changes to the code to make the test pass, even without understanding why the test now passes. (Sorry, it happens.)

Having a high density of testing code relative to the delivered code can actually increase the number of bugs in the delivered code!

To fix this:
- Test at a coarser level of granularity—perhaps at some level of integration that makes business sense. That reduces the testing code mass relative to the delivered code mass.
- Make sure that tests and code are independently developed.
- Don’t neglect system testing.

We tell our developers the story of Ada compiler development in the 1980s. It was a test-driven project driven by an acceptance suite. Most suppliers used the test suite to validate their compilers. Not Nokia. Nokia verified it against the Ada language definition. The Nokia compiler was the only Ada compiler not to exhibit the rendezvous bug that had been institutionalized in the test suite. There is currently a similar problem in the g++ test suite, in which a colleague of ours recently found more than forty bugs.

The software world is starting to learn this. As of 2005, one software notable would adopt a very conventional position and say, “Having the system-level tests before you begin implementation simplifies design, reduces stress, and improves feedback.”

Over time, the system evolves and the team eventually works out the bugs. The system goes to its first customer. Things go pretty well for the first two releases, though the competition has been making good inroads into the market. But the feature velocity is slowing because it’s difficult to turn around change quickly. To check in code, the unit tests have to pass. It now takes twenty-five minutes to run the tests. Projecting into the next release, we see that the time will soon grow to an hour. In general, the ceiling is unbounded.

Writing tests at the granularity of classes and member functions will create a large code mass whose maintenance will reduce your feature velocity, and for which the test times can grow without bound. In some companies we have seen these test suites grow to eight hours.

The solutions:
- Again, test at a coarser level of granularity. Systematically developed tests can give good coverage of business functionality and can adequately exercise code, even from a small test mass applied at fairly high-level interfaces.
- If a given piece of code cannot be exercised from such a high-level interface, then remove it. It has no bearing on anything visible to any external constituencies.

As a follow-on to long check-in times, the team decides to review current tests. They find that many of them—dozens, in fact—are now obsolete. Some of the test failures should be ignored, and some developers set some Maven flags to ignore those test results. The good news is that the team actually removes a few of the bad tests. The bad news is that they don’t create any new tests to cover the same code. The even worse news is that some of the test failures that were flagged as innocuous mask real bugs that the team will find later.

After the system goes to the third customer, customers start complaining about degrading usability. The complaints seem to fall into two categories.

The first complaint is straightforward: There was a new field on one of the forms, and when users tab rapidly from field to field, the habits they developed for a common case now caused the cursor to fall one field short of where it should be. The team couldn’t even understand the problem when it was faced with it. As recounted by Kent Beck, in the spirit of Smalltalk testing, the team found changes in the interface to be only annoying details; in the spirit of agile, it is “working code” that matters. Team members also had tested this interface; they clicked on individual fields with the mouse and entered the relevant data, finding that the system presented the right answer when they pressed “Enter.” The business logic was further validated by tests at a lower level.

TDD can give a false sense of confidence that the system is tested. Furthermore, it pre-ordains a certain ordering of events; doing the combinatorics is difficult and is rarely tackled by developers.

The solutions:
- Do system testing—or at least test at the level of meaningful business functionality—and usability testing. Treat TDD as if it had no validation or verification value—which is approximately true.
- Tell system testers what they need to test. System testers have long used to their benefit well-structured use cases that don’t constrain testers with overly detailed descrip-
tions designed to “help” the programmer. Use cases are a stronger alternative than user stories.

- Use systematic testing techniques to reduce redundancy among test cases.

The second complaint is a bit more problematic. The user interface proves to be not as productive as the competition’s, in spite of the fact that in the last release, a good usability person worked with development to improve the interface. Upon further study, it’s shown that the productivity lapse comes from a long pause in operator input when faced with non-repetitive operations and that there is a lot of rework that is caused by user errors. What is going on is that the entities known to the interface are a bad match for the mental model—both conscious and unconscious—of the end-user.

We approach the developers, and they insist that they have taken our advice from last time and that they have developed the system from tests that derive directly from user desires. We have a look at the architecture. We see the layered structure that came out of their bottom-up approach. True to the tests, the main organizing principle is around procedures. The business objects didn’t factor into the design, because design—true to TDD—is driven by tests. Combine this perspective with YAGNI (You Aren’t Going to Need It) and you have a perfect recipe for ignoring broad questions of structure: It’s all about functionality.

The GUI objects, of course, reflect the business objects of the underlying design, in concert with Model-View-Controller principles. These principles go to the heart of object-orientation (see the StickyNotes for references), directly engaging the end-user with the code. That the classes came about from pieced-together functions, rather than from broad domain thinking, leaves the system structure (and the GUI) beyond the user’s comprehension. Unfortunately, the competition has a sound user-conceptual model of the business, and those objects shine through in the screens, entities, and operations of the GUI. Faced with a new situation, the competition’s users are more likely to do the right thing than are our users.

A design driven by tests is necessarily a design driven by procedural thinking. It encourages procedural layering, rather than creating conceptually cohesive objects. The best that even a masterful interface can do is to capture these objects that badly reflect the user’s mental model.

To fix this:

- Do a little up-front design, as described previously.
- The architecture of a system is visible to the user in the way it works. Kent Beck once said that a good interface can’t hide a bad design, and Trygve Reenskaug has said that the architecture is key to a good human-computer interface.

Now team members think they finally are in good shape. They are eager to go into their fourth release, which is supposed to support a wide range of new functionality. Again and again they find the same kind of crosscutting problems as they had observed in the very early problems with user stories interfering with each other. Everything seems to require coordination between a large number of objects, and the developers get the feeling that everything is becoming global. We have a look at the changes they are making: None of them are well encapsulated or localized, because their functional decomposition structure cuts across the functional decomposition structure inherent in the architecture. The architecture has not been partitioned according to basic business structures, which would support evolution gracefully. By this time, the competition has gained so much market share that it can afford to buy out our company—and it does. The competition talks to the consultants and decides that a major training effort is in order. Our team has many things to learn, the last of which is:

Over time, the same TDD practices that drive a design away from supporting usability also drive it away from architectural soundness.

And again, the solution is:

- Don’t drive your architecture with testing. Drive it with lightweight domain analysis and enough up-front design that you can be confident about the structure of the GUI and the maintainability of the architecture.
- Start with abstract base classes and flesh them out as use cases come along.

So, now we’re working for the competition, and we gain some insight into what it’s doing right. It is following most of the suggestions above, which boil down to some simple principles:

- Do lightweight, up-front architecture.
- Use use cases, not user stories.
- Drive tests from use cases rather than from developer intuition. Test developer code against independently developed tests as soon as both are available.

They’re pretty simple principles. We find that our competitor has a lower bug density than we do. And, in fact, it has a strongly disciplined manner of class design. Once its classes are designed, using input from domain experts and usability folks, the competitor attaches pre- and post-conditions to the methods and asserts class invariants for each instance. This sharpens its thinking in a much more formal way than TDD ever was able to do. To this end, the team was using macros; in Eiffel or in Spec#, you find them directly supported in the language. The developers told the boss that this was a good idea. First, most tests only reach in and evaluate results for a specific, small set of values. Assertions can be set up to express entire sets of values or value ranges or arbitrarily complex business rule invariants that must always be true. System testing not only provides inputs and looks for correct outputs, but it also exercises all of these built-in asser-
tions that look for additional invariants that derive indirectly from the requirements. The developers thought that it would be a good idea to leave the assertions in the code in the field, either to drive recovery action or to print out a message that the customer should contact customer service to report a bug. But the boss was uncomfortable with this and told the developers not to do it. We don’t want our code failing in the field; it should continue to run. Even when admonished that continued operation in light of a failed invariant renders the program’s results invalid and that it would be better to stop the program than to silently corrupt the client data, the boss stood his ground. The overtaking company’s management discovered his reaction and re-assigned him to staff a telephone in customer service.

Exhaustive testing is impossible, and you cannot test quality into a product. Forward-engineered spot checks do not constitute adequate testing; in fact, their value is very low.

Instead:

- Use invariants and assertions that formally tie to the user needs as they are understood. While total formal program proving is a long way off, it is better to get the broader coverage that invariants and assertions can give than to use the spot-checking of a unit test.
- Don’t forget code inspections—one of the most cost-effective bug-spotting techniques we’ve got.
- As noted by Laurie Williams, if you’re pairing, you already have four eyes looking at every line of code, and that will help your quality immensely. Change pairings frequently so you don’t get into mind-lock with each other. And remember that even pair programming is no substitute for good design.

We know that this story has been about TDD and that TDD is not a testing method, but a design method—that it is the main testing focus on many projects to the contrary notwithstanding. But just in case you were interested in testing …

Don’t Neglect System Testing

Unit testing can help, and tests can play a role in sharpening one’s thinking—but not as most people practice TDD.

There is hope. Those who live at the leading edge of the curve with healthy skepticism and long hindsight are offering healthy alternatives. Many of the recommendations from above can be found embodied in behavior-driven development (BDD) as advocated by Dan North. BDD is a more sober and sane approach that integrates testing and development—not as a way to supplant careful structuring, but as an audit on how the system meets end-user needs. We are seeing other sound ideas that return to solid principles of quality-oriented development, such as Bertrand Meyer’s contract-driven development. The knee-jerk reactions that took place in response to overly heavyweight interpretation of their principles are extreme but are done absent responsible reasoning. Instead, follow the proven practices as described in these articles. They can be summarized as follows: Testing is an integral part of software development and requires the same careful attention to systematic design as writing the code itself. Just like the code, the tests must be designed; however, test design is essentially different from code design. One cannot simply augment cowboy coding with cowboy testing and expect to produce a quality product. *(end)*

Gertrud Bjørnvig is an independent consultant focusing on agile software development. During her twenty years in software development, she has been working at Scandinavian companies such as TietoEnator and Navision. As a use case expert, she has worked closely with test specialists and usability specialists to find a common ground in use cases that can support improved quality as well as improved usability in the final product.

**References**

Jim Coplien (http://users.rcn.com/coplien) has been a software professional for more than thirty years and is currently senior agile coach and systems architect at Nordija A/S in Copenhagen, Denmark. He is author of pioneering software design books including Advanced C++, Multi-paradigm Design, and Organizational Patterns of Agile Software Development. He is a founding member and Member Emeritus of the Hillside Group, the group that launched the software pattern discipline. When he grows up, he wants to be an anthropologist. Email Jim at JOCoplien@cs.com.

Neil Harrison is an assistant professor of computer science at Utah Valley State College in Orem, Utah. Before that, Neil developed software professionally for more than twenty years. He is the co-author of Organizational Patterns of Agile Software Development. Neil is acknowledged as the world’s leading expert on pattern shepherding and the pattern conference (PLoP) shepherding award is named after him. Email Neil at harrinse@uvsc.edu.

The authors thank Stine Laforce, Rex Black, Jeff Sutherland, Håkan Reis, and Magnus Mårtensson for comments, criticisms, and encouragement.

**Sticky Note**

For more on the following topic go to www.StickyMinds.com/bettersoftware.

**www.StickyMinds.com DECEMBER 2007 BETTER SOFTWARE 37**