A Story about User Stories and Test-Driven Development

Gertrud Bjørnvig, Gertrud & Co., Espergærde, Denmark
James O. Coplien, Nordija A/S, Kongens Lyngby, Denmark
Neil Harrison, Utah Valley University, Orem, Utah, USA

Welcome to TDD

Test-Driven Development, or TDD, is a term used for a popular collection of development techniques in wide use in the Agile community. While *testing* is part of its name, and though it includes tests, and though it fits in that part of the life cycle usually ascribed to unit testing activity, TDD pundits universally insist that it is not a testing technique, but rather a technique that helps one focus one's *design* thinking. The idea is that you write your tests first, and your code second. In this article, we explore some subtle pitfalls of TDD that have come out of our experience, our consultancy on real projects (all of the conjectured problems are things we have actually seen in practice), and a bit of that rare commodity called common sense.

A Conjectural and Metaphorical History of TDD

We want to start this story with a conjecture. It is not intended to represent truth but, in a popular Agile tradition, is a foundation for metaphor and allegory to help make some of the TDD problems more memorable. If elements of the story happen to be true, so much the better.

Once upon a time, software development was doing more or less O.K. when it employed Use Cases to help bridge the connection between developers and end users. Healthy organizations developed Use Cases for their intended audiences—including the coders and testers—all the while keeping them readable by the client. Use Cases were intended to avoid over-constraining the solution: they were written in terms of pre- and post-conditions, rather than attempting to convey the intent in a code-centric IF-THEN-ELSE style. Alistair Cockburn created an informal instrument to vividly present a single example of a Use Case enactment called *User Stories*, which had a true element of *story* to them. They were never intended to survive into the project.[1]

However, some organizations used Use Cases in the worst possible way, specifying details of program task sequencing instead of sticking to

[1] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001, page 17.

requirements. For example, there might be a business rule stating that one cannot go into a normal Danish train ticket office and place an order for train travel to Sweden, and another rule stating that such tickets can still be printed at main Danish ticket offices. The organization would create Use Cases for purchasing a Danish ticket and printing it at a Danish ticket kiosk, as well as exotic extensions whereby one calls the Swedish train company and books a ticket by phone and has it printed at a main Danish ticket office. The developers would turn this into a nested if-then-else hell, rather than writing the normal sequence, delineating special cases with extensions and business rules.

On the other hand, the Extreme Programming (XP) people, wanting to take things to the opposite extreme, turned the knob up to 10 on simplicity. Use Cases gave way to Alistair's User Stories—about 5 on the knob—which were further pared down to feature lists—all the way up to 10. Wanting to capture the market interest of those who formerly had employed Use Cases, the XP people took Alistair's name for his Agile articulation of a user interaction with the system. The XP feature lists came to be named User Stories and Alistair went in search of another name for his concept. An XP User Story was nothing more than "a mnemonic token of an ongoing conversation about a requirement."[2] Meanwhile, Alistair's concept—renamed to *Usage Narrative*—provided a good starting point for exploring the business space, a good stepping stone to more detailed Use Cases with enough information to support actual development. This concept would thrive in successful organizations as an application of the pattern *Narrative and Generic*.[3]

At about the same time, the Smalltalk community found that it was weak on testing. Part of this problem owed to the fact that most Smalltalk projects were too small to justify hiring a tester, so the only staff left to do testing were developers—who usually lacked professional testing skills. This gave birth to a testing methodology using a testing framework.[4] As the XP world moved away from Smalltalk and towards Java, the approach spawned a tool called JUnit. However, as time went on, the community found that unit testing didn't find many bugs—many still leaked through to the customer. System testing never was a big part of the culture, and in general it became difficult to sell testing to the XP constituency—testing was usually left to second-class citizens or was outsourced and, as such, was never a revenue-generating business. It didn't help that testing wasn't "fun," and that "fun" is a core value of XP. The TDD

---

[2] Agile Principles, Patterns and Practices in C#, Robert C. Martin and Micah Martin, Prentice Hall, 2007, page 14.

[3] Gertrud Bjørnvig, Proceedings of EuroPLoP '03, "Patterns for the Role of Use Cases," page 886.

[4] See the original paper by Kent Beck at www.xprogramming.com/testfram.htm, as published in the Smalltalk Report, October 1994.

advocates motivate its value by telling us that TDD is fun—not so much for its value, but for novelty for its own sake.[5] The testing framework and JUnit weren't garnering consulting gigs any more.

Further turning the knob up to 10, the XP people chose to simplify the complexity of up-front design. Ron Jeffries tells us:

> Get a few people together and spend a few minutes sketching out the design. Ten minutes is ideal—half an hour should be the most time you spend to do this. After that, the best thing to do is to let the code participate in the design session—move to the machine and start typing in code.[6]

Architecture was replaced in concept by the smaller and more abstract notion of *metaphor*, and even metaphor was limited to a bit of up-front work and the occasional reference during development as needed.

The approach left much to be desired. Programmers found that they needed a written record of some of the finer details of scenarios—much finer than what a User Story conveyed. They needed these in part just to help them understand requirements, but also needed them to write their tests. At the same time, while XP criticized heavyweight up-front design, it offered no real alternative except to "let the code do the talking."

One brilliant idea would address all of these problems: to re-cast the JUnit test scripting as a design activity rather than as a testing activity. This made the central concepts of XP more appealing to programmers than any testing regimen would, because design is *fun*. Development would necessarily start with JUnit: you had to write a test before writing any code. At the same time it appeased the collective conscience of the community by having a testing technique (now to publicly be called a design technique) in its vocabulary, addressing the weakness uncovered in 1994. XP could now point to TDD as its design methodology—something close to the programmer and far from the heavyweight architects as well as the second-class-citizen testers. It made sure that TDD would be portrayed that way through repeated insistence that TDD is *not* a testing method, but a design method. And in fact, it was touted even as an *analysis* method whereby we "put the analysis efforts into the tests."[7] Yet it was still offered as the safety net that allows programmers to move boldly forward by ensuring they don't step outside the boundaries of stipulated behavior—which means to employ the scripts as tests.

---

[5] Jimmy Nilsson, "Why is TDD so Boring?" www.jnsk.se/weblog/posts/tddisfun.htm, weblog of 22 March 2007, accessed 10 July 2007.

[6] Ron Jeffries et al., *XP Installed*, Addison-Wesley, 2000, page 70.

[7] Steve Freeman, "Test Driven Development: How do we know when we're done?" QCon, QE-II Conference Center, London, 15 March 2007.

A Journey with some Metaphorical Developers

The XP approach as it converged over the years appears to balance all of the forces: the need to engage the development community as well as the user community, the need to capture low-level scenario information, a sustained market for JUnit and the need to avoid heavyweight techniques such as up-front architecture. But in solving what were supposedly serious problems, is it possible that these changes actually made trouble? You've heard the tutorials; you've read the books. The three of us, in our work together with many clients, have seen where the rubber meets the road, and have had the opportunity to follow TDD projects over the course of several months (or, sometimes, years) to assess the long-term damage. Let's follow our metaphorical project around for a while to elucidate possible TDD pitfalls.

So the XP programmer came to work one morning, found a list of User Stories, and paired up with a colleague who had been working in the same area, and the two decided to code a certain User Story that day. They discussed between them how the feature should work and discovered they had to write two test cases to cover the User Story. They wrote the tests and, true to the TDD methodology, executed them before writing any code to ensure that they would fail. They did. It was simple — it took about ten minutes, well within a single Ron Jeffries design sprint.

The same pair then coded up the class. This, as it turns out, took a bit more work—about an hour of iterating the code to pass the tests. Along the way the pair created two more tests. Predictably, when they were finished, all four tests passed.

Here we find our first risk:

If the same people write both the tests and the code from the same base understanding or formalism, the chances of creating a self-fulfilling prophecy are high.

Here's the problem: Most software bugs reflect assumptions of the coder rather than an incorrect conveyance of those assumptions in code. A single programmer has a single set of assumptions, and any traps within such assumptions will not be caught if the same programmer writes the code and the test. This can happen even in simple cases, such as creating a linked list. Our programmers didn't know that their train reservation library would be used in a multi-threaded system, with the potential for one thread to be traversing a list of scheduling options while another application might be updating routing information in real

time. Eventually, this error would cause a member of the National Assembly to miss getting a train into the city for an important vote on a conflict with Sweden which would turn the political tide and ultimately lead to World War III.

Even two programmers who are pairing can mislead each other into assumptions that would be broken by at least one of a separated pair. To put it differently, pair design and pair programming lead to pair assumptions—a variation on groupthink. To remedy this pitfall, use the following techniques:

- ✓ **Develop the test from a different set of formalisms than are used for the design. For example, derive the design from a good Use Case description, while deriving the test from business rules or invariants.**

- ✓ **Have a different person develop the tests than the one who develops the code.**

You can use either or both of these techniques together.

The team develops a bit further and is considering another set of User Stories. The pair can see that these User Stories have detailed cases that interfere with assumptions made in a previous test case. In particular, they have found that a previous test case presumed that a given reservation could be found in the database using a ticket kiosk ID or main office ID as a primary key. The primary key captures the site making the original reservation of any ticket issued in Denmark. However, the new User Story offers that the end user can reserve a ticket to Sweden by calling the Swedish train company on the phone, and then issue the ticket at a main Danish train company location—even though it is impossible to directly make a booking to Sweden at a main Danish train office.

This leads to our second risk:

-  **User Stories (promises for a future discussion between a developer and a customer), processed and elaborated in isolation during coding, fail to capture the interesting extensions that come about from feature combinations or exceptional conditions. There is nothing to encourage the coder to think about these at TDD time: indeed, because many of these extensions involve multiple objects, such considerations are likely never to arise at the class level.**

To fix this, we again go back to the future:

- ✓ **Develop and iterate Use Cases up front, giving good coverage to all of the cases relevant to your next customer deliverable, and capturing other pre- and post-conditions that you know. Give particular attention to Use Case extensions and exceptions so you understand boundary cases. While the code may ultimately be the design, it is totally inappropriate as a tool to support the social activities of design. Use Cases, on the other hand, support a culture of dialogue with the client.**

As you write the Use Cases you will discover conflicts — conflicts even at the business level. A classic example from telephony is that you develop the call waiting feature, and then the call forwarding feature. If you develop either one independently, you miss facing the decision of what to do when a client is on a call, has call forwarding enabled but also subscribes to call waiting—and someone attempts to place a call to that client. If you capture this scenario information in JUnit or Cactus, one User Story at a time, you risk major rework at some point along the way—not only of code, but of the tests as well. It is a design issue that affects the most fundamental structures of the code—structures you can't refactor into the code if you miss them the first time around. Even worse, and perhaps more likely, is that your end user won't discover requirements bugs until deployment—bugs that could have been caught by clarifying scenarios before a single line of code is written. You can improve the stability of both your code and of your tests by using Use Cases as a kind of "fixed point" of development.

The developers hear our suggestion and they go back and re-do the original reservation User Story—including re-writing the test (and then it fails) and updating the code (the test still fails) and going into several iterations with test and code until both fit each other (that is, the test passes) and they seem to fit the modified understanding of the functionality (which was never captured in a test before the decisions were reduced to code.) The code ends being a mess because of these missed assumptions. No problem: they'll re-factor their way out of it. However, the refactoring browser isn't much help because most of the re-structuring at this point is taking place across class hierarchies. Well, we still tell the boss it's re-factoring and after a small one-week delay (on a two-week sprint), we're in good shape. In the mean time, our competitor has shipped another release.

This leads us to the third risk:

-  **Lack of good architectural delineation from up-front design and Use Cases can lead to major re-structuring events as the team learns, a bit at**

a time, what the system structure is supposed to look like. Re-factoring often isn't an option, sometimes because the changes go beyond what good re-factoring can achieve and take us into the arbitrarily general world of hacking and re-design.

The solution:

- ✓ **Do just enough up-front lightweight architecture to support the structuring of the development organization, to map on to the physical system architecture, and to do a good job of supporting usability. One important consideration is that the structure of the code fit the structure of the human/computer interface, so the development team should take cues from usability people when formulating the system object structure**.

The team checks out recent empirical findings by Maria Siniaalto and Pekka Abrahamsson and is relieved to find that they have done exactly the right thing.[8] Our developers found the same thing that Maria and Pekka did: that the use of TDD makes key architectural indicators worse. That runs counter to common claims that TDD leads to more cohesive systems. Though local complexity metrics are a bit better with TDD than without, the dependency management metrics were clearly worse. TDD is an architectural nightmare. But, architecture in hand, our adventuresome developers are again back on track.

It's finally time to demo the system to the customer, and the customer sits down and enters an unforeseen illegal sequence of keys. The customer turned out to be a good tester, because the program crashed. The sequence entered by the customer-as-user put the system into a state that had been unforeseen by the developers in fleshing out the user stories. It was certainly not a scenario the customer would have thought up beforehand; exhaustively enumerating error cases is, well, *exhausting*. The problem was in the interaction between two objects rather than in any single object.

Now we come to the fourth risk:

- ☠ **Testing individual classes leads to no guarantees about the soundness of the objects of that class interacting with the objects of another,**

---

8 Siniaalto and Abrahamsson, *Comparative Case Study on the Effect of Test-Driven Development on Program Design and Test Coverage*, ESEM 2007.

independently tested class—or even between cooperating objects of that single class.

This problem, too, can be resolved:

- ✓ **Write tests at the level of some meaningful business concept. The test should be something traceable to customer desiderata and possible system-user interactions, not to something inside the system.**

- ✓ **Tests should be written directly from customer input rather than from something created by the developer in isolation. Too many tests are created from the programmer's internal understanding of the program, rather than from an understanding of business needs—which are often emergent properties resulting from the interaction of several classes. There is rarely a direct mapping from the systems thinking of solving user/business problems to the structure of a simple, individual class. Only in those systems where such mapping is straightforward do simple techniques work. Few systems of interesting commercial value are so simple.**

The project takes our advice and in fact re-does everything so that the tests now reflect concerns of business value. The team has built a collection of individually robust objects that all pass their tests. A few of them have mocks, but the team has been smart to build higher-level constructs on lower level ones. True to TDD, they do this one test at a time, each one driving some functionality of some object, which in turn may depend on other objects below it.

So the team has struggled to get the first iteration to market. The industry knows that first iterations are easy: almost anything works on a greenfield project undertaken by a single small team. But Agile talks about change, and changes happen. Agile doesn't talk about bugs, and bugs happen. What are the repercussions of change and bugs on a TDD system? How about the propagation of change through the *test* code? How about the architecture's ability to encapsulate change when the structure rests on the execution paths of the first release? Will our team succeed, or will they continue to learn their way out of these problems? Things have only yet begun to be interesting.

Reflecting on Where We Are So Far

So far, we have introduced a small team using Test-Driven Development as their design method. Our metaphorical project struggled to deliver a product, and that

many of the struggles owed to TDD itself. These are not small bugs 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 inordinate rework. It leads to very spotty coverage of the design space, ignoring 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 within the past year, and our story reflects practices that they advise.[9] 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 their software to the field. We congratulate them. Let's pick up the story where we left off.

Change Happens

Another customer comes along with a killer bug. The development team goes in and tries to find it, yet the system is doing exactly what it is supposed to be doing: all the tests pass, yet the system doesn't work right. After three days of head-scratching the team finds that, in fact, one of the tests is wrong: just a coding error in the Java code for the test. Woops. Once the test is fixed, it fails. It turns out that this is a fairly common case, and that the word on the street had in fact been that at many TDD companies the motto was: "We don't fix bugs, we fix tests."

Here come the fifth and sixth risk:

In a typical TDD setting, half the code mass comprises unit tests. Those tests are code. Where there is code, there are bugs.[10] Programmers are psychologically programmed to trust their tests over their code—even though it takes only five minutes to write the test and an order of

---

[9] For example: Steve Freeman, "Test Driven Development: How do we know when we're done?" QCon, QE-II Conference Center, London, 15 March 2007; Jimmy Nilsson, "TDD, That's the way," Øredev 2006, Malmö, Sweden, 15 November 2006; and, Mary Poppendieck, "Stop-the-Line Quality," Øresund Agile, Copenhagen, 13 June, 2007.

[10] See, for example, Matt Stephens and Doug Rosenberg, *Extreme Programming Refactored.*

magnitude longer to write the code. They sometime will make arbitrary changes to the code to make the test pass, even without understanding. (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 these problems:

- ✓ **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**.

In fact, we tell them 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: they 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 40 bugs.

The software world is starting to learn this. As of 2005 an Agile advocate and software notable would finally reverse his earlier position and say, "Having the system-level tests before you begin implementation simplifies design, reduces stress, and improves feedback."[11]

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 25 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.

---

11 See Kent Beck, *Extreme Programming Explained, 2nd Edition*, Pearson, 2005, page 28.

This brings us to the seventh risk:

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 8 hours.

The solutions?

- ✓ **Test at a coarser level of granularity. Professionally 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 are in fact now obsolete—in fact, dozens of them. 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 in fact mask real bugs that the team would 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 was straightforward: There was a new field on one of the forms, and that when users tab rapidly from field to field, the habits they developed for a common case now cause the cursor to fall one field short of where it should be. The team couldn't even understand the problem when they were faced with it. First, in the spirit of Smalltalk testing, they found changes in the interface to be only annoying details: in the spirit of Agile, it is "working code" that matters. Second, they had in fact 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.

We now see the eighth risk:

☠ **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 solution?

- ✓ **Do system testing—or at least test at the level of meaningful business functionality—and usability testing. Treat TDD as if it has no validation or verification value—which is approximately true.**

- ✓ **To do this, you need to tell system testers what they need to test. System testers have long used well-structured Use Cases to their benefit—Use Cases that don't constrain testers with overly detailed descriptions designed to "help" the programmer. Use Cases are a stronger alternative than User Stories.**

- ✓ **Use systematic testing techniques to reduce redundancy among test cases, to get the greatest testing benefit for the fewest number of tests.**

The second complaint is a bit more problematic. The user interface proves to be not as productive as the competition's—all of this in spite of the fact that in the last release, a good usability person worked with development to improve the interface as best as possible. A study shows 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 owes to user errors. What is going on is that the entities known to the interface are a bad match for the entities in the unconscious (and sometimes the consciousness) 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 desiderata. We have a look at the architecture. We see the layered structure that came out of their bottom-up approach. The main organizing principle, true to the tests, 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 ain't gonna need it", another famous Agile

rallying cry) 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—about directly engaging the end user with the code.[12] 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. The competition has a sound architecture of objects that capture the 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 our users are.

This leads us to the ninth risk:

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 mental model.

To fix this:

- ✓ **Do a little up-front design, as described above. 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 the team thinks 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. They find, again and again, 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,

---

[12] See Reenskaug, and Coplien, "Things your mother didn't tell you about architecture and GUIs," ROOTS 2007, 27 April 2007, Bergen, Norway.

which would support evolution gracefully. By this time, the competition has gained so much market share that they can afford to buy out our company. They do. They talk to the consultants who have been there and decide that a major training effort is in order. They have many risks they must avoid, another one 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 so that you can be confident about the structure of the GUI and the maintainability of the architecture. Start just 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 they're doing right. They are following most of the suggestions above. These boil down to some simple principles:

- ✓ **Do lightweight up-front architecture**

- ✓ **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**.

Pretty simple principles. We find that they have lower bug density than we do. And, in fact, they have a strongly disciplined manner of class design. Once their classes are designed, using input from domain experts and usability folks, they attach pre- and post-conditions to the methods and assert class invariants for each instance. To do so sharpens their thinking, and does so 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 assertions that look for additional invariants that derive indirectly from the requirements. The developers furthermore 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 this. 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.

This leads us to eleventh risk:

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

This is actually not a risk, since a risk is a bad thing that might happen. This is a constraint, a problem that always exists and which smart teams must solve. They can solve it as follows:

- ✓ **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 you 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.**

- ✓ **If you're pairing, you already have four eyes looking at every line of code, and that will help your quality immensely.[13] Change pairings**

[13] See, for example, **Laurie Williams,** ***Pair Programming Illuminated.***

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 article 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 not withstanding. 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, and certainly not as the consultants teach it in the conference tutorials.

Some Final Lessons to be Learned

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—a few more, a few less—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.[14]

We are seeing other sound ideas that return to solid principles of quality-oriented development, such as Bertrand Meyer's Contract-Driven Development.[15] 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.

---

[14] See Dan North, "BDD – The Agile Evolution," Expo-C, Gothenburg, Sweden, 24 April, 2007.

[15] See Bertrand Meyer, **Øresund Complete**, Malmö, Sweden, 10 May, 2007.

Acknowledgements

Many thanks to Stine Laforce, Rex Black, Jeff Sutherland, Håkan Reis, and Magnus Mårtensson for comments, criticisms, and encouragement.