Introduction to Test Driven Development (TDD)

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation (Martin, Newkirk, and Kess 2003). In other words, it’s one way to think through your design before your write your functional code. Another view is that TDD is a programming technique. As Ron Jeffries likes to say, the goal of TDD is to write clean code that works. I think that there is merit in both arguments although I leave it for you to decide.

Table of Contents

1. What is TDD?
2. TDD and traditional testing
3. TDD and documentation
4. Test-driven database development
5. TDD and Agile Model-Driven Development (AMDD)
6. Why TDD?
7. Myths and misconceptions
8. Summary
9. Tools

1. What is TDD?

The steps of test first development (TFD) are overviewed in the UML activity diagram of Figure 1. The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).

Figure 1. The Steps of test-first development (TFD).
I like to describe TDD with this simple formula:

\[ \text{TDD} = \text{TFD} + \text{refactoring}. \]

TDD completely turns traditional development around. Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn’t present. In fact, they refuse to add even a single line of code until a test exists for it. Once the test is in place they then do the work required to ensure that the test suite now passes (your new code may
break several existing tests as well as the new one). Once your code works, you then refactor it to ensure that it's remains of high quality. This sounds simple in principle, but when you are first learning to take a TDD approach it proves require great discipline because it is easy to “slip” and write functional code without first writing a new test. One of the advantages of pair programming (Williams and Kessler 2002) is that your pair helps you to stay on track.

An underlying assumption of TDD is that you have a unit-testing framework available to you. Agile software developers often use the xUnit family of open source tools, such as JUnit or VBUnit, although commercial tools are also viable options. Without such tools TDD is virtually impossible. Figure 2 presents a UML state chart diagram for how people typically work with the xUnit tools. This diagram was suggested to me by Keith Ray.

Figure 2. Testing via the xUnit Framework.

Kent Beck, who popularized TDD in eXtreme Programming (XP) (Beck 2000), defines two simple rules for TDD (Beck 2003). First, you should write new business code only when an automated test has failed. Second, you should eliminate any duplication that you find. Beck explains how these two simple rules generate complex individual and group behavior:

- You design organically, with the running code providing feedback between decisions.
- You write your own tests because you can't wait 20 times per day for someone else to write them for you.
- Your development environment must provide rapid response to small changes (e.g. you need a fast compiler and regression test suite).
- Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too).

For developers, the implication is that they need to learn how to write effective unit tests. Beck's experience is that good unit tests:

- Run fast (they have short setups, run times, and break downs).
- Run in isolation (you should be able to reorder them).
- Use data that makes them easy to read and to understand.
- Use real data (e.g. copies of production data) when they need to.

http://www.agiledata.org/essays/tdd.html
2. TDD and Traditional Testing

TDD is primarily a programming technique with a side effect of ensuring that your source code is thoroughly unit tested. However, there is more to testing than this. You'll still need to consider traditional testing techniques such as functional testing, user acceptance testing, system integration testing, and so on. Much of this testing can also be done early in your project if you choose to do so (and you should). In fact, in XP the acceptance tests for a user story are specified by the project stakeholder(s) either before or in parallel to the code being written, giving stakeholders the confidence that the system does in fact meet their requirements.

With traditional testing a successful test finds one or more defects. It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that your system actually meets the requirements defined for it, that your system actually works and therefore you can proceed with confidence.

As with traditional testing, the greater the risk profile of the system the more thorough your tests need to be. With both traditional testing and TDD you aren't striving for perfection, instead you are testing to the importance of the system. To paraphrase Agile Modeling (AM), you should “test with a purpose” and know why you are testing something and to what level it needs to be tested. An interesting side effect of TDD is that you achieve 100% coverage test – every single line of code is tested – something that traditional testing doesn’t guarantee (although it does recommend it). In general I think it’s fairly safe to say that TDD results in significantly better code testing than do traditional techniques.

3. TDD and Documentation

Like it or not most programmers don’t read the written documentation for a system, instead they prefer to work with the code. And there’s nothing wrong with this. When trying to understand a class or operation most programmers will first look for sample code that already invokes it. Well-written unit tests do exactly this – the provide a working specification of your functional code – and as a result unit tests effectively become a significant portion of your technical documentation. The implication is that the expectations of the pro-documentation crowd need to reflect this reality.

Similarly, acceptance tests can form an important part of your requirements documentation. This makes a lot of sense when you stop and think about it. Your acceptance tests define exactly what your stakeholders expect of your system, therefore they specify your critical requirements.

Are tests sufficient documentation? Very likely not, but they do form an important part of it. For example, you are likely to find that you still need user, system overview, operations, and support documentation. You may even find that you require summary documentation overviewing the business process that your system supports. When you approach documentation with an open mind, I suspect that you will find that these two types of tests cover the majority of your documentation needs for developers and business stakeholders.

4. Test-Driven Database Development

If it’s worth building, it’s worth testing.
If it’s not worth testing, why are you wasting your time working on it?
At the time of this writing an important question being asked within the agile community is “can TDD work for data-oriented development?” When you look at the process depicted in Figure 1 it is important to note that none of the steps specify object programming languages, such as Java or C#, even though those are the environments TDD is typically used in. Why couldn’t you write a test before making a change to your database schema? Why couldn’t you make the change, run the tests, and refactor your schema as required? It seems to me that you only need to choose to work this way.

My guess is that in the near term database TDD won’t work as smoothly as application TDD. The first challenge is tool support. Although unit-testing tools, such as DBUnit, are now available they are still an emerging technology at the time of this writing. Some DBAs are improving the quality of the testing they doing, but I haven’t yet seen anyone take a TDD approach to database development. One challenge is that unit testing tools are still not well accepted within the data community, although that is changing, so my expectation is that over the next few years database TDD will grow. Second, the concept of evolutionary development is new to many data professionals and as a result the motivation to take a TDD approach has yet to take hold. This issue affects the nature of the tools available to data professionals – because a serial mindset still dominates within the traditional data community most tools do not support evolutionary development. My hope is that tool vendors will catch on to this shift in paradigm, but my expectation is that we’ll need to develop open source tools instead. Third, my experience is that most people who do data-oriented work seem to prefer a model-driven, and not a test-driven approach. One cause of this is likely because a test-driven approach hasn’t been widely considered until now, another reason might be that many data professionals are likely visual thinkers and therefore prefer a modeling-driven approach.

5. TDD and Agile Model-Driven Development (AMDD)

How does TDD compare with model-driven development (MDD), or more to the point agile model-driven development (AMDD) (the lifecycle for which is captured in Figure 3)? I believe:

- TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop.
- TDD provides detailed specification (tests) whereas AMDD can provide traditional specifications (data models).
- TDD promotes the development of high-quality code whereas AMDD promotes high-quality communication with your stakeholders and other developers.
- TDD provides concrete evidence that your software works whereas AMDD supports your team, including stakeholders, in working toward a common understanding.
- TDD “speaks” to programmers whereas AMDD speaks to data professionals.
- TDD is provides very finely grained concrete feedback on the order of minutes whereas AMDD enables verbal feedback on the order minutes (concrete feedback requires developers to follow the practice Prove It With Code and thus becomes dependent on non-AM techniques).
- TDD helps to ensure that your design is clean by focusing on creation of operations that are callable and testable whereas AMDD provides an opportunity to think through larger design/architectural issues before you code.
- TDD is non-visualy oriented whereas AMDD is visually oriented.
- Both techniques are new to traditional developers and therefore may be threatening to them.
- Both techniques support evolutionary development.

Figure 3. The Agile Model Driven Development (AMDD) lifecycle.
Which approach should you take? The answer depends on your, and your teammates, cognitive preferences. Some people are primarily "visual thinkers", also called spatial thinkers, and they may prefer to think things through via drawing. Other people are primarily text oriented, non-visual or non-spatial thinkers, who don't work well with drawings and therefore they may prefer a TDD approach. Of course most people land somewhere in the middle of these two extremes and as a result they prefer to use each technique when it makes the most sense. In short, the answer is to use the two techniques together so as to gain the advantages of both.

How do you combine the two approaches? AMDD should be used to create models with your project stakeholders to help explore their requirements and then to explore those requirements sufficiently in architectural and design models (often simple sketches). TDD should be used as a critical part of your build efforts to ensure that you develop clean, working code. The end result is that you will have a high-quality, working system that meets the actual needs of your project stakeholders.

6. Why TDD?

A significant advantage of TDD is that it enables you to take small steps when writing software. This is a practice that I have promoted for years because it is far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.
I think Bob Martin says it well “The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function”.

The first reaction that many people have to agile techniques is that they're ok for small projects, perhaps involving a handful of people for several months, but that they wouldn't work for "real" projects that are much larger. That's simply not true. Beck (2003) reports working on a Smalltalk system taking a completely test-driven approach which took 4 years and 40 person years of effort, resulting in 250,000 lines of functional code and 250,000 lines of test code. There are 4000 tests running in under 20 minutes, with the full suite being run several times a day. Although there are larger systems out there, I've personally worked on systems where several hundred person years of effort were involved, it is clear that TDD works for good-sized systems.

7. Myths and Misconceptions

There are several common myths and misconceptions which people have regarding TDD which I would like to clear up if possible. Table 1 lists these myths and describes the reality.

Table 1. Addressing the myths and misconceptions surrounding TDD.

<table>
<thead>
<tr>
<th>Myth</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>You create a 100% regression test suite</td>
<td>Although this sounds like a good goal, and it is, it unfortunately isn't realistic for several reasons:</td>
</tr>
<tr>
<td></td>
<td>• I may have some reusable components/frameworks/... which I've downloaded or purchased which do not come with a test suite, nor perhaps even with source code. Although I can, and often do, create black-box tests which validate the interface of the component these tests won't completely validate the component.</td>
</tr>
<tr>
<td></td>
<td>• The user interface is really hard to test. Although user interface testing tools do in fact exist, not everyone owns them and sometimes they are difficult to use. A common strategy is to not automate user interface testing but instead to hope that user testing efforts cover this important aspect of your system. Not an ideal approach, but still a common one.</td>
</tr>
<tr>
<td></td>
<td>• Some developers on the team may not have adequate testing skills.</td>
</tr>
<tr>
<td></td>
<td>• Database regression testing is a fairly new concept and not yet well supported by tools.</td>
</tr>
<tr>
<td></td>
<td>• I may be working on a legacy system and may not yet have gotten around to writing the tests for some of the legacy functionality.</td>
</tr>
<tr>
<td>The unit tests form 100% of your design specification</td>
<td>People new to agile software development, or people claiming to be agile but who really aren't, or perhaps people who have never been involved with an actual agile project, will sometimes say this. The reality is that the unit test form a fair bit of the design specification, similarly acceptance tests form a fair bit of your requirements specification, but there's more to it than this. As Figure 3 indicates, agilists do in fact model and document for that matter, it's just that we're very smart about how we do it. Because you think about the production code before you write it, you effectively perform detailed design as I highly suggest reading my Single Source Information: An Agile Practice for Effective Documentation article.</td>
</tr>
<tr>
<td>You only need to unit test</td>
<td>For all but the simplest systems this is completely false. The agile community is very clear about the need for acceptance testing, user testing, system integration testing, and a host of other testing techniques.</td>
</tr>
<tr>
<td></td>
<td>This is partly true, although easy to overcome. Scalability issues include:</td>
</tr>
<tr>
<td></td>
<td>1. Your test suite takes too long to run. This is a common problem with a equally common solutions. First, separate your test suite into two components. One test suite contains the tests for the new functionality that you're currently working on, the other test suite contains all tests. You run the first test suite regularly, migrating older tests for mature portions of your production code to the overall test suite as appropriate.</td>
</tr>
</tbody>
</table>
8. Summary

Test-driven development (TDD) is a development technique where you must first write a test that fails before you write new functional code. TDD is being quickly adopted by agile software developers for development of application source code and is even being adopted by Agile DBAs for database development. TDD should be seen as complementary to Agile Model Driven Development (AMDD) approaches and the two can and should be used together. TDD does not replace traditional testing, instead it defines a proven way to ensure effective unit testing. A side effect of TDD is that the resulting tests are working examples for invoking the code, thereby providing a working specification for the code. My experience is that TDD works incredibly well in practice and it is something that all software developers should consider adopting.

9. Tools

The following is a representative list of TDD tools available to you. Please email me with suggestions.

- cppUnit
- csUnit (.Net)
- CUnit
- DUnit (Delphi)
- DBUnit
- JUnit
- NDbUnit
- OUnit
- PHPUnit
- PyUnit (Python)
- NUnit
- Test::Unit (Ruby)
- VBUnit

10. References and Suggested Online Readings

- Agile Data Modeling
- Agile Database Best Practices
- Agile Model Driven Development (AMDD)
- Comparing the Various Approaches to Modeling in Software Development
- A Roadmap for Testing RDBMSs
- Single Source Information: An Agile Practice for Effective Documentation
- Test Driven Development: A Practical Guide by Dave Astels
- Test Driven Development: By Example by Kent Beck
This book describes the philosophies and skills required for developers and database administrators to work together effectively on project teams following evolutionary software processes such as Extreme Programming (XP), the Rational Unified Process (RUP), the Agile Unified Process (AUP), Feature Driven Development (FDD), Dynamic System Development Method (DSDM), or The Enterprise Unified Process (EUP). In March 2004 it won a Jolt Productivity award.

This book describes, in detail, how to refactor a database schema to improve its design. The first section of the book overviews the fundamentals evolutionary database techniques in general and of database refactoring in detail. More importantly it presents strategies for implementing and deploying database refactoring, in the context of both "simple" single application databases and in "complex" multi-application databases. The second section, the majority of the book, is a database refactoring reference catalog. It describes over 60 database refactorings, presenting data models overviewing each refactoring and the code to implement it.

This book presents a full-lifecycle, agile model driven development (AMDD) approach to software development. It is one of the few books which covers both object-oriented and data-oriented development in a comprehensive and coherent manner. Techniques the book covers include Agile Modeling (AM), Full Lifecycle Object-Oriented Testing (FLOOT), over 30 modeling techniques, agile database techniques, refactoring, and test driven development (TDD). If you want to gain the skills required to build mission-critical applications in an agile manner, this is the book for you.

11. Translations

- Chinese

12. Let Me Help

I actively work with clients around the world to improve their information technology (IT) practices as both a mentor/coach and trainer. A full description of what I do, and how to contact me, can be found here.