

Putting Test-Driven Development into Practice

By Jimmy Nilsson
www.jnsk.se/weblog/
2004-04-21

Introduction

Test-Driven Development (or TDD) is about writing tests before writing the real code. In doing this, the tests will drive your design and programming.

My aim with the paper is to provide you with thoughts and ideas about TDD so you'll want to try it out for yourselves. For those of you who are already familiar with TDD, you'll get some new ideas to help you become even more efficient.

This paper has been written with the primary purpose of assisting my presentation "Putting Test-Driven Development into Practice" at VS Live. I might transform the paper into an article at a later date.

Agenda

The paper is loosely structured in three parts. The first part discusses why TDD should be used and what it is. The second part discusses how TDD is used, with a real world demo, and the third part discusses miscellaneous concepts and ideas. Let's get started with part 1.

Why Test-Driven Development?

TDD sounds dull and boring, and developers often expect it to be a pain in the backside. They couldn't be more wrong! In my experience, the opposite is true - it's actually great fun, which came as a surprise to me. I guess the reason that it is such fun is that you get instant feedback on your changes and since we are professionals, we enjoy creating high quality applications. I think TDD is love!
:-)

Another way to put it is that TDD isn't about testing... It's about programming and design. It's about writing simpler, clearer, and more robust code!

TDD is yet another technique to add to your toolbox. Are you convinced yet? If not, have a look at my rationale.

Why? Improved quality

The reason I started with TDD in the first place was that I wanted to improve the quality of my projects. Improvement in quality is probably the most obvious and important effect.

We don't want to create applications that crash when the customer uses them for the first time or applications that break down when we need to enhance them. It's just not OK anymore.

On several occasions I have delivered applications and the customers have found bugs. Perhaps the same thing has happened to you too. If you say it hasn't, some of you would be lying to me, or perhaps suffering from memory loss!
:-)

TDD won't automatically help you never release products with bugs again, but the quality will improve.

The effect of improved quality you can get by writing tests *after* the real code. What I mean is that you don't have to apply TDD (which means writing tests *before* the real code), you just need a lot of discipline. On the other hand, using TDD gets the tests written. Otherwise, there is a very great risk that you won't write any tests when you're pressed for time, as always happens when it gets to a late stage in the projects. Again, TDD makes the tests happen.

Why? Improved simplicity of design

The second effect you can expect when applying TDD is to see improved simplicity of design. In the words of two popular sayings, "Simple is beautiful" and "KISS" (Keep It Simple S...). They are very important because, for example, complexity produces bugs.

Instead of creating loads of advanced blueprints covering every little detail upfront, when using TDD you will focus on the core customer requirements and just add the stuff the customer needs. You get more of a customer perspective than a technical perspective.

In the past I've been *very* good at overcomplicating simple things... TDD helps me keep focused and not do anything other than what is really necessary *now*.

This effect (getting improved simplicity of design) requires TDD. It's not enough to just write the tests afterwards.

Why? Improved productivity

Yet another effect of TDD is that you will get high productivity, all the way.

This might sound counterintuitive at first. When you start a new project, it feels very productive to get going and write the real code. At first you are very productive, but it's *very* common that the productivity completely drops near the end of the project.

Bugs start cropping up, the customer decides on a couple of pretty big changes that make everything unstable, you find out that you have misunderstood some things... Well, you get the picture.

An old Swedish saying goes: "Ropa inte hej förrän du är över bäcken!". It means something like, "It ain't over till you've crossed the river".

BTW, you shouldn't ask the customer if you should use TDD or not, at least not if you're asking for more payment/time/whatever at the same time. He will just tell you to do it right instead.

:-)

When considering the project from start to finish, if using TDD incurs no extra cost I believe you should just go ahead. The customer will be happy afterwards when he gets the quality he expects.

A colleague of mine has been *very* sceptical of the need for automatic unit tests (created before or after real code). He told me that during his two decade-long career automatic unit tests would not have helped him once. However, I think he changed his mind a bit recently. We were working together on a project where he wrote a COM component in C++ and I wrote tests as specifications and as just tests. When we were done, the customer changed one thing in the requirements. My colleague made a small change, but the tests caught a bug that occurred just four times in 1000 iterations. The bug was found after only seconds of testing, compared to hours if it had been done manually. And if it had been done manually, the bug would most probably not have been found at all, but would have shown itself during production.

What is Test Driven Development?

Now I have tried to get you motivated into starting with TDD, let's have a closer look at how the process or flow is. We'll get back to this in a few minutes when we will investigate it in a bit more depth with a real world demo.

First of all, you start writing a test. You make the test fail, for the reason that you should be sure that the test is testing what you think. This is a simple and important rule. Even so, I have skipped over it several times and that is just asking for trouble.

The second step is to write the simplest possible code that make the test pass.

The third step is to refactor if necessary, and then you start all over again, adding another test.

If we use NUnit lingo, the first step should give you red light/bar, the second step should give you green light/bar.

I have to admit that I often fall back into the old habit of doing upfront design. However, thinking about the problem in different ways is often the most efficient thing to do. A little bit top-down, a little bit bottom-up.

What is Refactoring?

I mentioned refactoring above as the third step in the general process of TDD and I think I should briefly explain the term. Refactoring is making small well-known changes, step by step in order to improve the design of existing code. That is, to improve its maintainability without changing its observed behavior. Another way to say it is to change *how*, not *what*.

So, you don't have to come up with a perfect design up front. That is good news, because it can't be done anyway.

:-)

Let's take a look at a simple example: Encapsulate Field. Assume you have a public member like this:

```
public string Name;
```

When you apply Encapsulate Field, you just change this to using a property instead. Perhaps you want to intercept the setting of the name, or (as happens below) the value should just be able to read, not set.

```
private string _name;
public Name
{
    get
    {
        return _name;
    }
}
```

Something that is worth mentioning is that the Whidbey-release of Visual Studio .NET (Visual Studio .NET 2005) will include support for a couple of refactorings so that you can apply them in a very productive way.

In order to be able to use refactoring in a safe way, you *must* carry out extensive tests. Otherwise you will introduce bugs and/or you will prioritise not making any changes simply for the sake of maintainability, because the risk of introducing bugs is just too large. And when you stop making changes because of maintainability, your code has slowly started to degrade.

What are the tools?

I've often heard developers say that they can't use automatic unit testing because they don't have any good tools. Well, the mindset is way more important than the tools, although tools do help, of course.

I wrote my own tool (see Figure 1) a couple of years ago and used it for registering tests of sprocs, COM components and .NET classes. Thanks to this tool, I could skip those forms with 97 buttons that have to be pressed in a certain order in order to execute the tests.

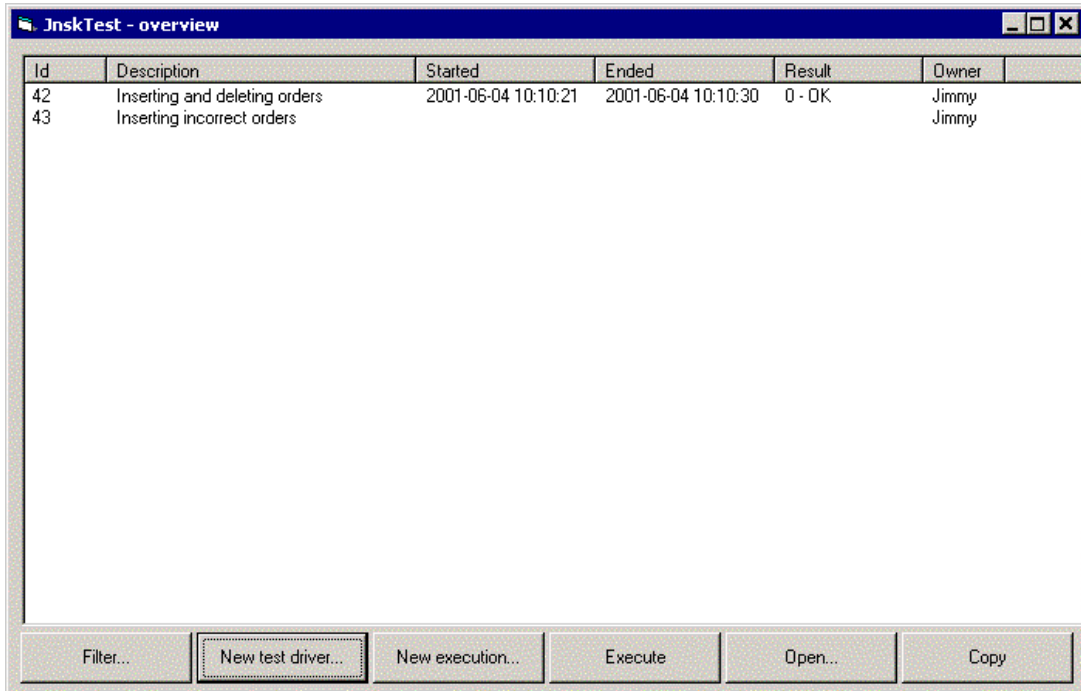


Figure 1 *JnskTest.*

Later on, when NUnit was released (see Figure 2), I started using that tool instead. Honestly, using NUnit is *way* more productive, it takes five minutes to learn and ten minutes to get addicted.
:-)

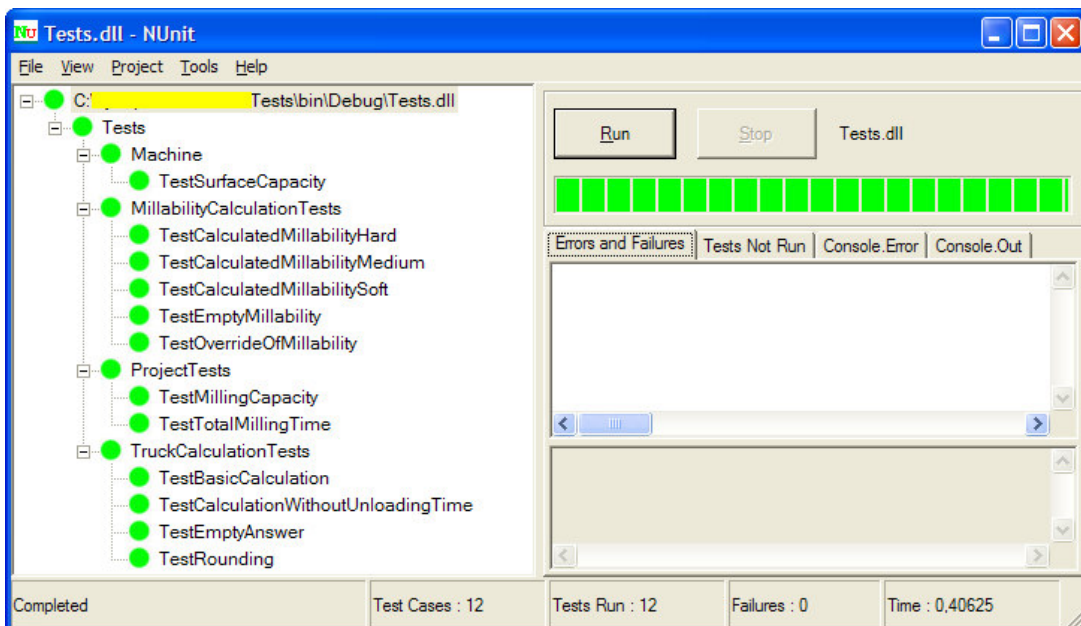


Figure 2 *NUnit, the GUI-version.*

That's enough talk, now it's time to show NUnit in action in a real world demo.

How? Time for a loong demo :-)

I'm assuming that you are familiar with NUnit so instead I 'll focus on how to apply TDD.

I know, I know. There is an enormous amount of good demonstration texts on how to use TDD, see the books in the reference section for example. Anyway, I'd like to have a go at demonstrating it myself. Rather than giving what is most common, I won't use a homemade toy example, but I'll use an example from a real world application of mine.

I was asked by Dynapac to write an application that they can bundle with a new line of Planers that they have produced. The planers are used for removing old asphalt before paving out new. Asphalt milling is used to restore the surface of asphalt pavement to a specified profile. Bumps, ruts, and other surface irregularities are removed, leaving a uniform textured surface.

The application should be used for calculating a number of different things, helping to optimise the usage of the planers. The application should be able to calculate how hard the old asphalt is, how long it will take to plane a certain project, how many trucks are needed to keep continue milling with the minimum cost, and lots of other things.

I'm going to use this real-world application to demonstrate how TDD can be used so you get a feeling for the flow. I'm going to focus on calculating the number of trucks. See Figure 3 below to see a planer and a truck in action, planing off old asphalt.



Figure 3 *A planer and a truck in action, planing off old asphalt.*

It is important to have the correct number of trucks for transporting the old asphalt. If you have too few, the planer will have to stop every now and then or move slower. If you have too many, it's a waste of trucks and personnel. In both cases, the cost will increase for the project.

OK, let's do this the TDD way. I start out by writing three different sections in a text file like this:

```
Functionality
- Calculate millability
- Calculate milling capacity
- Calculate number of trucks needed

Test

Refactorings
```

As I said, I'm going to focus on the calculation for the number of trucks now, so I put a marker in the file on that line so I know what I decided to start with. As soon as I (or more usually the customer) come up with another piece of functionality that I don't want to work on right now, I add it to the file. The same goes for tests and refactorings. So that text file will help me not forget about anything, while still being able to focus on one thing at a time.

The next thing to do is to think about tests for the truck calculation. At first I thought it was very simple, but when I started to think about it a bit, I found out it was actually pretty complex. Not *extremely* complex, but complex.

So let's start out easy and add a simple test. Let the test check that when no input is given, zero trucks should be needed. First of all, I create a new project that I call Tests. I add a class that I call TruckCalculationTests and I do the necessary preparations for making it a test fixture according to NUnit, such as setting a reference to NUnit.Framework, adding a using clause (not necessary, but nice), and decorating the class with [TestFixture]. Then I write a first test, which looks like this:

```
[Test]
public void TestForZeroAsResultWhenNoInput()
{
    TruckCalculation theCalculation = new TruckCalculation();

    Assert.AreEqual(0, theCalculation.NeededNumberOfTrucks);
}
```

As a matter of fact, when writing that extremely simple test, I actually made a couple of small design decisions. First, I decided that I need a class called TruckCalculation. Secondly, I decided that that class should have a property called NeededNumberOfTrucks.

Of course, the test-project won't compile yet since we haven't written the code that it tests so let's continue with adding the "real" project, so to speak. I add another project that I call PlanPave.Domain. In that project I add a class like this:

```
public class TruckCalculation
{
    public int NeededNumberOfTrucks
    {
```

```

        get
        {
            return -1;
        }
    }
}

```

What is probably a bit “irritating” is that I fake the return value to -1. The reason is to force a failure for my test. Remember, we should always start with an unsuccessful test, and for this specific test just returning zero would clearly not fail.

Then we go back to the Tests project and set a reference to the PlanPave.Domain project. We also add a using-clause, and build the whole solution. After that we open the Tests project (the project file or the DLL) in NUnit, and execute the tests (or actually only the *test* now). Hopefully we get a red bar. Then we go back to the property code and change it so it returns zero instead, back to NUnit and a green bar.

Is there anything to refactor? Well, I’m pretty happy with the code so far. It’s the simplest code I can think of right now that satisfies the tests (or – again – the *test*).

OK, we have taken a small step in the right direction. We have started working on the new functionality for calculating number of trucks, and we have a first simple little test that gives us pretty good code coverage so far.

:-)

Let’s take another small step. Again, we take the step by letting tests drive our progress. So, we need to come up with another test. One thing is that I need to show both a rounded result and a more exact result, at least to one decimal. The rounded result must always be rounded up since it’s hard to create a half truck. (I know what you are thinking, but it’s not in the customer requirements to deal with a mix of different sized trucks. Ah, it’s great to make up, I mean, tell you the requirements as we go<g>.) That’s a decent and necessary test, but I don’t feel like dealing with it now. I want to take a more interesting step, so I add the rounding test to the text file under the Tests header.

Instead, I’d like to take a step with the calculation. Heck, it can’t be so hard to calculate this. I have to take transportation distance into account, as well as transportation speed, truck capacity, milling capacity, unloading time, loading time, and probably a couple of other things. Moving simply on, let’s say that I don’t care about transportation for now, nor the time for loading. I won’t even care about other factors that are as yet unknown to me. (Pretty wild, eh?) The only things I care about now are milling capacity, truck capacity and unloading. So, things are simple enough at the moment. I can write a test like this:

```

[Test]
public void TestResultWhenOnlyCapacityIsDealtWith()
{
    TruckCalculation theCalculation = new TruckCalculation();

    theCalculation.TruckCapacity = 5;
    theCalculation.MillingCapacity = 20;
    theCalculation.UnloadingTime = 30;
}

```

```
    Assert.AreEqual(2, theCalculation.NeededNumberOfTrucks);
}
```

What I did above was assume the milling capacity was 20 tonne/h and each truck can deal with 5 tonne each time. I also said that unloading the truck takes 30 minutes, so each truck can only be used twice in one hour. That should mean that we need two trucks, so I test for that.

When I try to compile the Tests project, it fails since the new test expects three new properties. Let's add those properties to the TruckCalculation class. They could look like this:

```
public int TruckCapacity = 0;
public int MillingCapacity = 0;
public int UnloadingTime = 0;
```

Hmmm, that wasn't even properties, just public fields. We'll come back to discussing this later on. Now we can build the solution, and hopefully we now get a red bar in NUnit. Yep, expected, and wanted. We need to make a change to NeededNumberOfTrucks, to make a real (or, at least, more real) calculation.

```
public int NeededNumberOfTrucks
{
    get
    {
        return MillingCapacity /
            (TruckCapacity * 60 / UnloadingTime);
    }
}
```

Hmmm... Perhaps this was too big a leap to take here, but I'm feeling confident now. :-)

OK, build again and then over to NUnit. Green bar, oops, red. How can that be? Ah, it wasn't the last test that was the problem; that test runs successfully. It's the old test that now fails. I get an exception from it. Of course, the first test doesn't give any values to the TruckCapacity and UnloadingTime properties so I get a division by zero. A silly mistake to make, but I'm actually very happy about that red bar. Even that tiny little first test helped me by finding a bug just minutes (or even seconds) after I created the bug. I need to change the calculation a bit more:

```
public int NeededNumberOfTrucks
{
    get
    {
        if (TruckCapacity != 0 && UnloadingTime != 0)
            return MillingCapacity /
                (TruckCapacity * 60 / UnloadingTime);
        else
            return 0;
    }
}
```

Build, NUnit and green bar. Good, another step in the right direction, secured (at least to a certain degree) with tests.

Any refactorings? Well, I'm pretty sure several of you hate my usage of the public fields. I used to myself, but I have since changed my mind about them. As long as the fields should be both readable and writable and no interception is needed when reading or writing the values, the public fields are as good as properties, more or less. The good thing is that they are simpler; the bad thing is if you need to simply add a breakpoint when one of the values is set. I refactor that later, if and when necessary. Feel free to do it now if you want.
:-)

Another thing I could refactor is to add a [SetUp] method to the test class that instantiates a calculation member on the instance level. That's right, don't forget about your tests when you think about refactoring. Anyway, I can't say I feel a great urge for that change either, at least not right now.

Yet another thing that I'm not very happy about is the code for the calculation itself as it's a bit messy. I think a better solution would be to take away what I think will be the least common situation of zero values in a guard. In this way the ordinary and real calculation will be clearer. This change is not extremely important, more like a matter of personal taste. Anyway, let's make the change called Replace Nested Conditional with Guard Clauses:

```
public int NeededNumberOfTrucks
{
    get
    {
        if (TruckCapacity == 0 || UnloadingTime == 0)
            return 0;

        return MillingCapacity /
            (TruckCapacity * 60 / UnloadingTime);
    }
}
```

Build, NUnit. Still green bars, and the code is alright too. At least I think so right now. Well, an even better code snippet is probably to use Consolidate Conditional Expression and change the tests into a method instead, like this:

```
public int NeededNumberOfTrucks
{
    get
    {
        if (!_IsNotCalculatable())
            return 0;

        return MillingCapacity /
            (TruckCapacity * 60 / UnloadingTime);
    }
}
```

Now it's time to add another test. But I think this little start of writing a new class for calculating the needed number of trucks was enough to show the flow of TDD.

Just a quick note. What I showed you in the demo was pretty extreme so as to make the ideas obvious. For example, you won't benefit from IntelliSense if you write the tests before the methods or properties exist. If you think it is a problem, you could start with creating empty methods/properties first when you have decided that they are needed to be called from a test. On the other hand, what we are talking about here is the interface and it could be good to write it twice to get an extra check, as it were.

Design effects?

During the demo I said that when I wrote the tests, it was very much a design activity. Below I have listed some of the effects I have found when designing with TDD instead of upfront design:

- ***More client control***
Earlier I have thought that as much as possible should be hidden from the outside so that classes are configuring themselves, for example. To use TDD, you need to make it possible for the tests to set up the classes the way they need them. This is actually a good thing because the classes become much easier to reuse.
- ***More interfaces/factories***
In order to make it possible to use stubs or mock objects, you will find that you need to gather functionality via interfaces. If you come from a COM background, you will have learned the hard way that doing this has many other merits.
- ***More sub results exposed***
In order to be able to move a tiny step at a time, you need to expose sub results so that they are testable. (If I had continued the demo, you would have seen me expose more subresults for the transportation and loadtime, for example.) As long as the sub results are just read only, this is not such a big deal. Just be careful that you don't expose too much of your algorithms.
- ***More to the point***
If I had started with an upfront design for the demo example, I'm pretty sure that I would have invented a truck class holding on to loads of properties for trucks. Since I just focused on what was needed to make the tests run, I skipped the truck class completely. All closely related to a truck that was needed was a truck capacity property and I kept that property directly on the truck calculation class.
- ***Less Coupling***
For instance, the UI will more or less be forced not to intermingle with the domain model. The coupling is also greatly reduced thanks to the increased usage of interfaces.

When you use TDD the API has already been used once when the time comes for your real consumer to use it. Probably, the real consumer will find that the API is better and more stable than it would have been otherwise.

To conclude this section, test-friendly design (or testability) is a very important design goal nowadays.

Future maintenance

Maintenance goes like a dream when you have extensive test suites.

I know, I know. Up until now I've been sounding like a salesman, promising the moon and the stars, the end to all war, etc, etc. Of course there are some problems too, for instance, you also need to maintain the test code.

If you don't have extensive test suites when you need to make changes, you can go ahead and create tests afterwards. It's not much fun, and you probably won't be able to create high quality tests long after the code was written, but it's often a better approach than just making the changes and crossing your fingers. You then have tests when it's time for the next change, and the next.

In a forum I read recently, someone asked "Are there really any good excuses *not* to use TDD?" It might be that this forum was a bit biased, but the answer was "no". :-)

When a bug is found

Despite using TDD, bugs will appear in production code. When it happens and you are about to fix the bug, the process is very similar to the ordinary TDD one.

You start by writing a test that exposes the bug, then write the code that makes the tests (the new and all the old tests) execute successfully. Finally you refactor if necessary.

BTW, another way of seeing TDD is as an aid to developers. In the past I have tried hard to make the compiler help me by telling me when I do something wrong. Tests are like the next level up from that. The tests are pretty good at finding out what the compiler misses.

If you follow the ideas of TDD, you will find that you don't need to spend nearly as much time with the debugger as you would otherwise have to. Even so, there might be situations where you need to inspect the values of variables, and so on. You can use `Console.WriteLine`-calls and then find the output at the `Console.Output` tab in NUnit. Another thing you could do is attach to the `nunit-gui.exe`-process from the Visual Studio .NET IDE, and then your breakpoints will work as expected.

Speed tips

In the demo you saw that I was taking very small steps, but of course you don't have to do that all the time. You make tiny increments when you are unsure and big leaps when you are confident.

Compare it to when you find yourself in an unfamiliar place, where in order to find your way you use a map and you go very slowly, following the progress carefully on the map. OK, you prefer GPS, right?

:-)

But when you are on your way home to your parents on the highway, you go very fast without the help of any maps or anything. But if something makes you unsure, like making a diversion due to railroad work, again you need to slow down and move carefully.

I'd like to say one more thing about the importance of instant feedback. An error found five minutes after it was created is way easier and cheaper to solve than five days or five months later.

Isolation or Performance?

A common question about tests is that you should prioritise the tests to execute fast or in isolation. If given a choice, prefer isolated tests to highly efficient tests, otherwise one failing test will fail other tests and then it's hard to get the status, as well as it being harder to spot the problem.

Yet another problem with un-isolated tests is that they might have to be executed in a certain order. You definitely want to be able to execute one single test without having to execute some other test before!

Another important point is that it should be possible to run a test again and again with consistent results.

Problems?

I said earlier that I sounded like a salesman so I had to give a disadvantage of using TDD. There are of course other problems. Let's have a look at the following:

- ***UI***
It is hard to use TDD for the user interface. TDD fits easier and better with the domain model, the core logic, but this isn't necessarily a bad thing. It helps you be strict in splitting the UI from the domain model, which is a basic design best practice. Ensure you write very thin UI and factor out logic from the forms.
- ***Database***
Databases do cause a tricky problem with TDD because once you have written to the database in one test, the database is in another state when it's time for the next test. This causes trouble with isolation. You could write your code to

set up the database to a well known state between each test, but that adds to the tester's burden and the tests will run much more slowly.

You could also write tests so that they aren't affected by the changed state, which might reduce the power of the tests a bit. Yet another solution is to use stubs or mock objects to simulate the database apart from during the nightly tests when a backup of the database is restored before the tests.

Yet another way is to use transactions just for the sake of testing. Of course, if your tests are focusing on the transactions, then this isn't a useful solution.

- ***False sense of security***

Just because you get a green bar in NUnit doesn't mean that you have zero bugs - it just means that your tests haven't detected any bugs. Make sure that TDD doesn't lead you into a false sense of security. On the other hand, just because your tests aren't perfect doesn't mean that you shouldn't use them. They will still help you quite a lot with improving the quality!

- ***Losing the overview***

TDD is pretty much a bottom-up approach and you must be aware that you might lose the overview from time to time. It's definitely a very good idea to reengineer your code from time to time into UML, for example, so you get a bird's eye perspective. You will most certainly find out several refactorings that should be applied to prepare your code for the future.

Stubs/Mock Objects

To avoid having to write the code in a certain order, you can use stubs or mock objects for faking code that hasn't been written yet which will prevent dependencies in the tests.

Stubs or mock objects are also often used for increasing test execution speed. It's important that the test suite can execute within seconds. If, for example, database interaction means that the tests take minutes to run, then the tests won't be used for instant feedback, and some of the benefits are lost. A better approach would be to use stubs or mock-up objects for simulating the database, except when tests are executed at times for integration or at the nightly build or such.

Another common situation for when stubs or mock objects are used is to simulate a scarce resource. For instance, you may not get the possibility to connect to the mainframe during development.

White box/black box

A common question regarding TDD is whether or not to use white-box or black-box testing. Should you test private methods or just methods that are publicly available (such as methods in the interface)?

I prefer to use black-box testing and test only the public interface with my automatic unit tests. The negative effect of this is that the public interface often gets a bit larger. I usually need to expose sub results as property gets. The bigger interface is read-only, so it's not a major problem. Sure, there's a risk of showing too much of the algorithm, but the larger interface often proves useful for the real consumer.

I also use ordinary assertions (such as `System.Diagnostics.Trace.Assert`, or preferably a custom implementation that better integrates with NUnit, but that's another story) for pre- and post-conditions in my methods (similar to Design by Contract by Bertrand Meyer). Of course, those assertions will help when running the automatic unit tests too.

A friend of mine notified me that it's not a matter of black box since I know the inner workings of the public methods. His suggestion was to call it grey box.

Integration tests

A common problem is caused when only integration tests are created instead of first creating unit tests. Both kinds of tests are very useful, but help in different ways. Using TDD will make the tests happen.

You could also write automatic tests for the acceptance tests, of course. There is no need to let someone do manual work when it is easy to do automatic instead. By taking out routine tasks, the manual resources can focus on coming up with ways of how to extend the automatic tests instead in order to improve quality even more.

In this context, I'd also like to mention that developers often say that there's no point in testing their own code. Sure, it's likely that others will find errors that you wouldn't find by yourself (this is one of the points of reviewing.) but I still think it's our responsibility to remove as many bugs as possible, and this way the QA department (if you're lucky enough to have one) can concentrate on trickier stuff. And again, if you use automatic tests, it costs very little for you to run the tests after each and every change.

Documentation

The tests you write are high quality documentation. That is yet another good reason for writing them in the first place. Examples are really helpful in order to understand something and the tests are examples both of what should work and what shouldn't.

Another way of thinking about the unit tests is that you reveal your assumptions. And that goes for the other developers too. I mean, they reveal their assumptions on your class. If their tests fail, it might be because your class isn't behaving as is expected, and this is of course very valuable information to you.

Acceptance tests

A common phenomenon is to find errors in the test cases from the customer, and this has happened to me on several occasions. When you tell the customer about it, he

feels that you are a true professional, working so much with the acceptance tests that you understand them well enough to find errors.

Tests as deployment support

In my experience, many production time problems are caused because of differences in the production environment compared to what is expected. If you have an extensive set of tests, why not use them for checking the deployment of the finished application as well? Pay once, reap the benefits many times over :-)

Summary

As I said in the beginning, there are people who think TDD is dull and boring, and expect it to be a pain. Now you see that *they* are wrong, and *you* understand that TDD is simple and fun. TDD is true love!
:-)

So, to summarize all this, by using Test Driven Development, you will get:

- Improved quality
- Improved simplicity in design
- Improved productivity

Thanks for reading and good luck with your TDD experiments.

References

- Kent Beck: Test-Driven Development
- David Astels: Test-Driven Development
- Martin Fowler: Refactoring
- Robert C. Martin: Agile Software Development. Principles, Patterns, and Practices
- NUnit: <http://sourceforge.net/projects/nunit>