Programming Practices: Part 2 – Thoughts on TDD

MARCH 10, 2010

Brian Harry MS

Well, it seems the last post wasn't too controversial. Let me try something that might be a bit more controversial. Heck, it might even get some people down right agitated with me but that's OK, disagreement is a useful tool to drive clarity and understanding.

I don't like Test Driven Development. I don't just not like it, I think it's a bad idea.

How's that for an inflammatory statement? Probably gonna make the cool kids black ball me :)

I don't spend a bunch of time debating the fine points of the various popular development techniques with people so I suspect someone is going to jump in and tell me I've missed the point of TDD – and maybe they are right. Let me refine my statement a bit and then launch into both what I like about TDD and what I don't like. So here's a little softer statement.

I don't like a literal interpretation of TDD.

If I think about what the goals of TDD are (and I'm doing a bit of reverse engineering to get there), then here is what I like about TDD:

1) Focus first on how you are going to use a thing, not the thing – I watch many people early in their software development career (including myself) make the same mistake over and over. They have an application to build. They envision and architecture, decompose the components and then start building them. After a few years they start to realize that the contracts between the components are really important so they start to get very rigorous about encapsulation, abstraction and contracts but their overall approach is the same.

I can't tell you how many times in my life I've done this and spent hours or days building enough components to get reasonably high up in the dependency stack and then go to build the next layer on top (it might be the UI or it might be a service interface or just a higher level abstraction) and realize, crap, it's all wired wrong.

The API isn't really built the way I like, the components aren't factored quite right, etc. The result is that i have to write more code at that next layer up than I should have to and the code doesn't flow very well.

The way I now approach this problem is that I always start at the top level (or a very high level) layer in the system and work down. Specifically before I write any code for a class, API, layer, … I write one or more samples that use the API to to accomplish some purpose. This shows me what I want the flow to be an what the natural abstractions are for the typical use cases of the service that I'm building. After each sample, I go back and visit previous samples and refactor until I get all of the samples using a consistent API/abstraction, resulting in a minimal set of code and clear and easy to understand flow.

For example, when I created the framework I've built for managing asynchronous UI in Windows Forms, before I created a line of code, I took the WinForms designer and wrote 6 different dialogs. Each of them demonstrated some important characteristic or extensibility point I wanted to have. The first was to show the bare minimum code one would have to write to get a dialog that would load asynchronously, be cancelable and give the user appropriate feedback. The next showed that I could pass parameters to the background loading process, implement a refresh model, etc. Other samples demonstrated how to coordinate the work of multiple background threads, different ways of displaying progress, different ways of notifying the background thread that it has been canceled, etc. All the dialogs were fully written (and rewritten several times) before I wrote a line of code on the async framework. While I didn't cover every part of the API, I had a very clear picture of the API/contract/abstractions that I wanted and could begin to conceptualize the design of the async framework. Of course, as I built the framework, I learned even more and went back a few times and refactored my samples but the samples always guided the work.

In a sense, you can think of these samples as test cases – in fact I often use them that way. Sounds a little like TDD, right? It kind of is. It's also got some characteristics of Behavior Driven Development (BDD) but in a little bit I'll tell why I see it as pretty different than a literal interpretation of TDD. My first vote of confidence in TDD though, is that I do believe it is a technique that can help you focus on how to use something first and later on what it does.

2) Don't write code you don't need (YAGNI) – YAGNI stands for You Ain't Gonna Need It. Another very common mistake I've seen in my career (and, again, made myself many times) is to try to imagine the ultimate end of where a program will go and build an architecture/implementation that sets you up to get there. Good developers are always thinking about all the cool new features/requirements they'd like to add down the road and they'd like to make sure they build their code in a way that enables it.

The problem is that, no matter how hard you try, you can't predict the future. You can sit here today and imagine what the requirements are going to be 6, 12, 24 months from now. You may believe you see very clearly where it should go but you don't. Please take my word for that. It's a very hard conversation to have with an eager developer who really wants to "build it right". Don't. Build what you need now and don't build more. Down the path lies lots of unused code, unnecessary abstractions, complexity that no body understands the need for, etc.

The issue is requirements change. I promise. 6 months from now you will look back and wonder what you were thinking. Because you'll have people actually using your app and they'll be giving you tons of feedback on stuff they want and it won't be the things you thought they'd want. I've lived it 100 times at least.

So does that mean that architecture doesn't matter and you should just put on blinders and build exactly what you need this very moment. No, I'm not quite saying that. Design your code with clear abstractions and generality in mind. Design it in a way that is extensible and composable. Just don't add requirements that you don't currently have. If you build a system with a clean, well factored architecture: encapsulation, separation of responsibility, clear contracts, etc, your code will be in a good position to tackle new requirements as they come. But assume no matter what you do, you're going to be doing some refactoring of that code when you get there. You might as well have less of it to refactor.

Back to TDD. I believe TDD helps you get here. It's a very rigorous focus on only writing the code that you can concretely identify that you need. I like that.

A note on extensibility – I was discussing this post with a colleague yesterday and he asked me to make sure I made a point about extensibility. Nearly all developers like to make their code extensible. However, many developers believe that their own implementation is "special" and has requirements that can't be met by the extensibility interfaces. Down that path is a bad extensibility model. If you can't build your own implementation in the same extensibility framework that you are providing for others to use, I can almost guarantee you they won't be able to use it either.

OK, so enough about what I like about TDD. Let's get to the juicy stuff – what don't I like:

1) I find it inefficient – If you read my last post then you know that I refactor like crazy. The idea of having unit tests that cover virtually every line of code that I've written that I have to refactor every time I refactor my code makes me shudder. Doing this way makes me take nearly twice as long as it would otherwise take and I don't feel like I get sufficient benefits from it.

Don't get me wrong, I'm a big fan of unit tests. I just prefer to write them after the code has stopped shaking a bit. In fact most of my early testing is "manual". Either I write a small UI on top of my service that allows me to plug in values and try it or

write some quick API tests that I throw away as soon as I have validated them. Once the code has started to settle, then I go back and write the unit tests that I'm going to use to help with regression testing down the road. I don't mean by this, that I wait until the app is done to build the tests but rather I do it iteratively at a component or requirement boundary.

2) Backing into an architecture – This is probably by biggest issue with a literal TDD interpretation. TDD says you never write a line of code without a failing test to show you need it. I find it leads developers down a dangerous path. Without any help from a methodology, I have met way too many developers in my life that "back into a solution". By this, I mean they write something, it mostly works and they discover a new requirement so they tack it on, and another and another and when they are done, they've got a monstrosity of special cases each designed to handle one specific scenario. There's way more code than there should be and it's way too complicated to understand.

I believe in finding general solutions to problems from which all the special cases naturally derive rather than building a solution of special cases. In my mind, to do this, you have to start by conceptualizing and coding the framework of the general algorithm. For me, that's a relatively monolithic exercise. When I've got the basic framework in place (I may still have parts of it still stubbed out), then I start testing to make sure all of the special cases are handled by the algorithm the way I pictured (and of course, as I said in my last post, I do this by stepping through all of my test cases in the debugger to see exactly what it is doing).

I'm not saying that it isn't possible to build a good architecture following TDD. I'm saying that lots of people are already inclined toward piecemeal architecture and I believe literal TDD enables these natural tendencies too much. I also find the constant moving back and forth between test case and code to be too distracting while I'm trying to hold the design in my head and get the core algorithm fleshed out.

Conclusion

I like many of the goals of TDD but I don't like some of the mechanism to get there. I suppose we'll see how many of you are avid TDD proponents and maybe we'll have vigorous debate about the pros, cons and interpretations of TDD. I look forward to it.

Till next time...

Brian

Zachary Spencer

TDD, like all practices, is a means to an end. If you have a good way to achieve that end without TDD then good on ya!

However, TDD is a means that people can grasp and learn. It's a means that can be applied in situations where there are serious environmental barriers to achieving those ends.

For instance, if you have a team of novices, mandating TDD will make it much, much easier for them to focus on how they will use the code simply because they must write a use for the code before they actually write the code itself.

If you have a team of experts, they should know more about when to and when not to write automated tests, and should be far better at determining how to write their code from a "how does this code get used?" perspective. Mandating TDD may not be as beneficial in that case.

Overall though, it's a pretty solid practice that has increased level of quality of code at my shop quite a bit. Do we do it all the time? No. Should we? That depends ;).

Mark B.

Brian, I think you've converted me. It never occurred to me to simply write the UI first - more so to treat the UI (or any higher level) as a test until the layer below it is dialed in. Top-Down-Development?

Gregory A. Beamer

TDD, and testing period, are scientific endeavors, as you are proving code works in a measurable way. Many developers are artists, as they have no proof that any of their software works and rely on QA, when there is a QA department, to be the scientists. Ouch!

I prefer a mixture of TDD and TED (test early development). When I am not sure I fully understand the problem, TDD helps me scientifically define the problem that has to be solved, as I can model a path through the software that acts how I feel it should act. I can then talk to the business people and state "as I see it, when we enter X, Y and Z, we get A back" and they can confirm or deny.

On the other hand, when I have a very good undestanding of the problem and have a good grasp on the issue at hand, I will often code, wrap in a test, Pex the solution for fringe cases, and leave it at that. I still have the benefits of proof, but I did not waste as much time. I know some purists that are cringing if they are reading this, but it is my bag, not theirs.

If you follow the above paradigm, you create some tests from code and some tests before code. If you find that all of your tests are early, rather than part of the design step, you have one of two conditions:

A) A bias against TDD

B) A job with few challenges

Just my two cents.

Peace and Grace,

Greg

Twitter: @gbworld

Ben

This part in particular struck me as strange:

"When I've got the basic framework in place (I may still have parts of it still stubbed out), then I start testing to make sure all of the special cases are handled by the algorithm the way I pictured (and of course, as I said in my last post, I do this by stepping through all of my test cases in the debugger to see exactly what it is doing)."

You're essentially designing and executing one-off tests to ensure that the behavior of your system has been specified correctly. The big downside to this is that neither you nor your collaborators can easily repeat the tests to ensure that a future modification has not broken the current behavior.

One big benefit of TDD is that you spend a whole lot less time in the debugger, because you've essentially done the equivalent work up front when specifying your system via executable tests. As a nice side effect, these "executable specifications" can be run in the future to facilitate refactoring.

Brian Harry MS

I'm not saying that I don't end up with reusable unit tests, I do. What I am saying is that a passing test doesn't mean that your code is doing what you think it is. I don't care if your tests are automated or not, you should run through all of them under the debugger and make sure the code is actually doing what you think it is. Correct result != correct code.

Brian

Realist

TDD seems to decently increase the amount of time required to develop a feature in an application, as well as the size of your code base. All code requires maintenance as changes are made, and this costs time and money.

What happens if your testing code has bugs? Do you need tests for your tests (who will police the police?).

It's all well and good if you are at a company that is profitable and can afford the time to follow this approach, but it can really hurt in a company with a low profit margin where the most important priority is the code the customer runs. Commercial realities dictate that feature complete and 98% bug free (on time) is better than feature complete and 99% bug free - 6 months late.

Whilst I do see some benefit to TDD, I don't understand the need some developers have to push it on other developers. A guy I once interviewed said "I can't program without writing tests first." Hello world!

Overall a thoughtful, well written piece. I look forward to more!

BacchusNaur

At my job, we have a rule. A programmer cannot perform QA on his or her own product. You've heard of this rule before. It's a common sense solution to a problem that plagues any programmer who is under any form of pressure be it time or quality or cost. Programmers under such pressure test until it works. They will code defensively if they are worth their salt, but that is not a given. Now, this isn't a result of laziness or malice. They just don't have time to think of every condition that could arise and they need to Get Things Done.

I would be weary to institute TDD in a pressure environment with a group of programmers who have never used it for fear that the test would just formalize the positive case testing that most programmers do. In my opinion, to adopt TDD, one must change one's way of looking at the code one produces to accept that it is fragile and will break. One must also take a leap of faith that the time invested in test design will pay off in maintenance cost.

Michael Ruminer

Largely in response to Zachary's comment as well as Brian's post in

genera I'll comment.

Not specific to TDD but TDD is one of the mechanisms where I seem to

most often encounter a significant 'mandate' in its use. I cringe at

the mandatory stance too often taken because I interpret that if I

must mandate to an entire set of people (in Zachary's example at least

junior people and potentially all people) then what I am mandating is

not working. Most developers want quality output and I should be able

to show and to lead so that they find for themselves TDD is valuable -

if indeed it is valuable for them or our team. If I have to perpetually mandate

it then #fail. Zachary started his comment with that caveat of TDD as one means to an end. And returning his good on ya' if it works well for his team.

Of course, we all mandate policies and opertions for example check in

comments or gated checkins or any of a hundred other things. I know

mandates.were not your intent but that word inevitably pops up and

makes my brain hurt whenever TDD adoption is in discussion. Even by someone as obviously pragmatic as Zachary.

Is it only my perception or have others too often found that if TDD

was in use it was under mandate to start, which is defensable, but

continued under mandate for much longer than it should if it were

providing the value?

I believe it can provide value but in the pure form of utilization I

see the ROI as the exception not the rule. I want exit criteria for

quality and how it is achieved is dependent on that persons best

method of achieving the quality coupled with teamwork and MOST

importantly mentoring that encourages methodologies that may work with

a huge dose of pragmatism.

Mike Brown

Just a thought on your two cons for TDD.

1. Refactoring shouldn't change the semantics of your code. If you write a test to the spec rather than to the line of code, an internal refactoring of the implementation shouldn't impact it. That's why I NEVER write tests against private/protected functions.

For large scale refactorings like moving namespaces, class/property/function names...well if you're not using Resharper or some other tool to help you, you're asking for trouble.

If you're changing the actual logic of the code under test, it's not a refactoring it's either a new feature or a bug fix, both of which should have a test written (to validate the feature or prevent a bug regression).

2. I agree that TDD done wrong can lead to a big ball of mud but to be honest if the developer can't avoid this with TDD, chances are he'll have the same results without. At least with TDD you have tests that give some hint to what the code is supposed to do.

David Douglass

To date I've been unwilling to criticize TDD due to lack of experience with it. I've have worked with the concepts using NUnit and so far I'm discouraged. Here's what I've found:

1) I'm spending more time writing unit tests than the software under test.

2) Most unit tests are a waste of time. If you have 100 functions, one of which has a bug, than the other 99 tests aren't really adding value. Of course, you don't which has the bug, so you're stuck writing all 100 tests.

3) Unit testing is an investment in the future. But many businesses are, for better or worse, only concerned with the short term. It's difficult to do unit testing unless there is an immediate payoff.

Isaac

Interesting post Brian :-) In response to your two criticisms of TDD - which can be very valid - I would say the following: -

1. Inefficiency - I've found that there's an art to writing unit tests almost as much as writing the code itself. The rules for writing the two are not always the same. I found myself writing lots of little helper methods with my unit tests in order to make them "less brittle" so that e.g. refactoring my main code wouldn't force a massive rewrite of my unit tests. We didn't do this initially, and treated our unit tests as "throwaway" code, and suffered for it. Treat it like a first class citizen, and it won't let you down :-)

2. Architecture - the one thing it drives you down the path (unless you "cheat" with something like Typemock) is that it encourages SRP and design by contract, where you have clear dependencies and responsibilities between classes - no bad thing in my book. There is always a risk of your code going in circles or spaghetti - but then,

that's a risk irrespective of TDD IMHO. The question is whether one is able to refactor their code after passing their last unit test to keep the code as clean as possible.

I am indeed a fan of TDD, but I also respect other who aren't - but at the same time, the majority of those I know who don't like it (and I'm not suggesting that you're in this group!!) haven't fully embraced TDD as a whole - including some of the nitty gritty bits like keeping your tests clean, refactoring them as well as your main code etc.

hardtosay

Since you seem confident that you write good code can you post a complete working example and documents for those of us interested in learning more?

Thanks.

Brian Harry MS

Mike Brown,

We may be using the term refactoring differently. To me refactoring is not only about reorganizing the code but it's about creating new abstractions/generalizations that didn't exist before. So when I refactor, I do often change the semantics of the code. I often get part way into it and realize that if I think about it a different way I can create a more general/clean solution to the problem and that refactoring often has significant ripple effects.

Brian

Brian Harry MS

hardtosay,

I do plan on posting some code a bit further into this series. However, if your goal is to find something wrong with it to discredit my advice, you will succeed. I'm not saying that my coding is flawless. I'm recommending some practices that I use that I believe lead to good design and good code. If you don't like them, you need not use them.

There is a bit of hubris involved anytime someone recommends anything. If I recommend a good restaurant to you, then I implicity saying "I know what good food is" and "I know what you'll like" and probably more. Maybe I do and maybe I don't. But I don't think fear of being wrong or having people disagree should discourage us from making recommendations when we feel like we have a valid reason for doing so.

Brian

Brian Harry MS

Maybe a bit further down this series, I'll write a post on the sins of automated testing. We do a lot of it at Microsoft and I've seen it go wrong a lot of ways.

Brian
