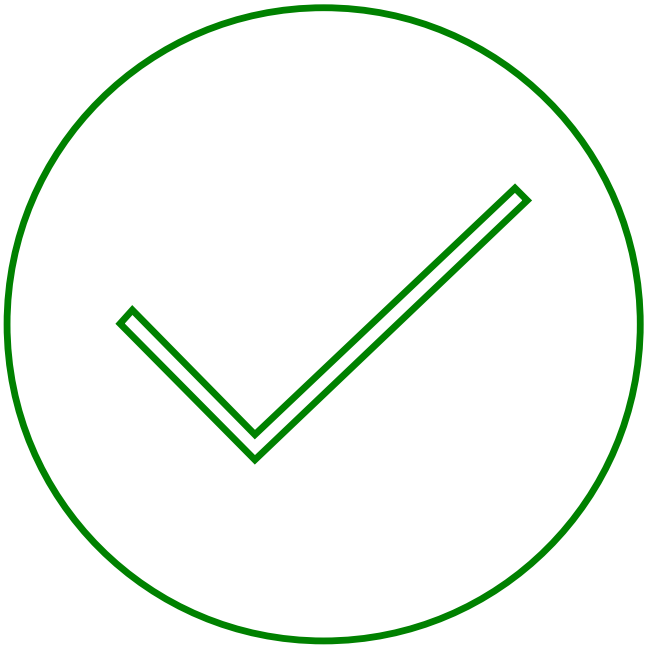

INTUITIVE TESTING

WITH

LEGACY ASP.NET MVC

BY NICK CHAMBERLAIN

Preface

When I started writing code professionally, my first project was someone else's first project. I inherited it from another junior developer who was also brand new to the game. It was a "safe" project for a junior developer because it only read data from various sources and put them on a web page for people in the business to look at and send to the customers.

I sat down with my new project to fix a bug - one of many bug-fixes I was assigned to in my junior role. I thought "what a mess." That's a knee jerk reaction... more of a "jerk" reaction. I was judging this developer who had, like me, just got out of college. His first assignment was this: "take this huge legacy application and completely rewrite it as a modern web application." It wasn't the same as my: "fix this bug" assignment.

Reading a couple books about design, SOLID, and testing doesn't make you ready for every task in your new junior developer job. Had I known this, I wouldn't have had this attitude of "man, what was this guy thinking?... why did he put so much logic in these controllers?... why didn't he use a Repository for this?... he's barely using MVC the right way..." What I would've said was: "wow, this guy didn't have a lot of time to deliver this project... he must not have had anyone to help him out... this is classic 'trial by fire'... poor guy."

Eventually, I was given a bigger assignment. This time I was right in that unfamiliar territory that my predecessor was in. I was going to be put in my place.

The task was this:

1. The app is an ASP.NET MVC 2 web application with about 10 controllers and 50 views
2. Every controller's actions *read* data from various sources including SQL and IBM UniData (o.g. document database)
3. International customers who are getting quotes for our products need information in their native language and currency (internationalization)
4. Make the MVC views for these quotes "editable"
5. Information on the quotes should be editable in place
6. If the MVC view is edited on the page, saving the page will write to a separate SQL database for later retrieval

I had to turn my comfortable, junior-safe, read-only application into a CRUD app. It looked like I needed JavaScript too. My first real *work* that wasn't bug fixes. It was both exciting and terrifying. I felt just like my friend did before me... a hard deadline, no experience, and smelly code.

What kinds of smells am I talking about?

- LINQ to SQL entity models first built for SQL access, then abandoned and only used as POCOs... keeping the LINQ to SQL-generated infrastructure code (unnecessary coupling)
- Scattered usage of Models, ViewData, and logic inside of controllers
- Models had single, 1000+ line methods containing data access and business logic code (single responsibility principle violation)
- CRUD was a mixture of both Repositories and LINQ to SQL code intermixed with connections “new’d” up throughout single controller methods (coupling to dependencies)

- Most of the beneficial layering provided by MVC was thrown out the window (untestability)

My issue was not with being able to read the code, or fix bugs in it, I'd already been doing that. It was with safely touching a larger piece of it for this feature.

- There was money on the line,
- There was a marketing team that chartered the project
- There was potential downtime
- There was my future as a junior developer on the line

At the time, the concept of unit testing felt out of my reach. Developers who were better than me did testing. Developers in big software companies with actual teams did testing. Only developers who wrote brand new SaaS apps in Silicon Valley did TDD.

I knew **how** to test. I'd watched some videos and read some articles about testing and TDD. I knew what mocks were. I knew how to run tests with a test runner. Most importantly, I knew **why** to test. For me, the reasons for **why** to test applied directly to my project. Once I moved past the *my career is at stake, oh my god* inner monologue, I started thinking about what I was going to do. There were more constraints:

- I couldn't find any place in the code where I could start writing tests
- I didn't understand that this particular app wasn't testable
- I needed to make this app testable
- I needed to make this app testable while delivering the feature
- I didn't have time to waste
- I was adding JavaScript to a mission critical part of a C# web app
- I was introducing a risky feature
- I was adding a feature that was ultimately customer-facing

- I needed to write some very reliable code
- I was getting bug-fix requests while I was writing this feature
- I needed to avoid adding to my bug fix backlog
- I needed to move on to other projects after I finished this feature
- I would get change requests after I delivered the final product

I had a strong gravitational force inside me saying *try writing tests for this*. Do full-scale TDD right? Wrong...

I studied up some more. I learned that Test Driven Development was great for writing new code. I found plenty of resources on writing failing tests, making them pass, moving on to the next. I learned different styles of TDD like BDD, “Detroit” style, and “London” style. I saw a dream of writing test and code in perfect rhythmic harmony. Then I looked back at my project.

I couldn't imagine applying TDD to this project. I couldn't visualize writing a single test for the project. The tests I had seen through reading and studying were for classes that had clear responsibilities and, more importantly, were decoupled enough so that they could be exercised in isolation. In my project, the areas that needed testing seemed completely inaccessible by outside code. I had no idea how I was going to get these components to a state where I could isolate them and write automated tests for them. I almost gave in to the cowboy coder way, trying to ignore all of those risks I was afraid of. Operating in *debug-and-hope-for-the-best* mode isn't something I get excited about.

In addition to it being obviously hard to test, I didn't know what to test. Everything?

- I could have tried to test everything
- I could have picked all the controllers and tried to test them thoroughly.

- I could have set a goal: 30% code coverage before I start writing any code.

None of these would have been effective. I had to be more intelligent and precise about my efforts. I had to ask myself different questions.

- How far into existing code should I test?
- How should I write tests for the new code I'm writing?
- How do I isolate existing code into testable pieces?
- How should I balance integration with unit tests?

There was a huge, intimidating gap between the testing I wanted to do and the testing I could do. I wanted to one day find the TDD bliss that practitioners spoke of. I only saw that bliss far into my future as a senior developer writing a brand new, shiny app from the ground up. The reality is that even senior developers inherit legacy code. They may write greenfield code, but it becomes legacy as soon as they deliver it. Having come across multiple, seemingly untestable apps, I can confirm that having the ability to adapt your testing strategy quickly will undoubtedly make you a better programmer.

What this Book is About

My goal for writing this book is simple. I want developers who are just starting with automated testing to bridge the gap between how to write unit tests and how to put unit tests to work in their daily coding life.

Merriam-webster defines "intuitive" as the power or faculty of attaining to direct knowledge or cognition without evident rational thought and inference[1]. As developers, we are demanded to act with intuition to solve complex problems by breaking them down into parts. We attack those parts that we can solve from intuition first. This frees up time for us to work on the parts that we can't solve with intuition.

I called this book "Intuitive Testing with Legacy ASP.NET MVC" because testing can be an activity that we approach with intuition. We gain intuition by practice.

It's understandable that training resources use sample applications where they are able to demonstrate key principles of testing greenfield apps. I believe, however, that the best way to gain intuition is to be challenged with difficult situations. Your first inherited legacy application will likely give you difficult testing situations. You will need testing intuition to be successful. Having a good strategy is going to make gaining and applying this intuition easier.

I'd like to present some techniques that will help you build the strategy for approaching your first few automated testing situations with a legacy application. A strategy that you can use to go from **knowing how to test** to:

- knowing what to test
- knowing what not to test
- knowing how much to test
- knowing how to design a test strategy

---

[1] "Intuitive". Merriam-Webster. Merriam-Webster, n.d. Web. 12 June 2016. http://www.merriam-webster.com/dictionary/intuitive

What this Book is Not About

This book is not about how to use a certain testing framework. I have chosen a stack for writing tests that I enjoy, and I have tried several others. The choice is up to you, and it's not a permanent choice. Structuring your test project and SUT the right way will help you prevent being locked into one choice for testing frameworks. I'd suggest not wasting much time on the choice and just choose one now, if you haven't already

It is also not about why you should unit test. I want to talk more about how to be sure you are focused on testing for the right reasons. I know that you are convinced that automated testing is good. It's important to make sure that you always have your eye on the prize with these benefits. I'd like to help you develop a strategy for staying focused on generating value with your tests.

I'm also not going to tell you what specific code you need to test in MVC or any other framework. Obviously, every application is different. Different hands have touched it and different techniques have been used to solve similar problems. To tell you to test a controller method a certain way isn't going to help you build a strategy when you run into a slightly different implementation of that Controller method.

Finally, I don't want to spend too much time defining TDD, BDD, and Integration/Unit/Regression/etc. testing. Those definitions are important and we may use them to frame our testing strategies. If we spend too much time on them, though, we may not reach our true objective - to start shipping tests. I want you to generate wins from this book. I want you to reap what you sow.

Who Should Read this Book

This book is primarily for people beginning, or wanting to begin using automated testing. It will also help anyone who knows the subject deeply, but is unable to move forward with using it in practice. This is the difficulty with a lot of programming practices. Learning a brand new subject in software isn't usually that hard. The hard part is applying it to what you do professionally.

I also believe that you lose skills that you don't use professionally. You may become an expert in a subject by studying and practicing it for years. It begins to deteriorate as soon as you stop using it professionally. This deterioration requires you to go back and apply some deliberate practice to refresh.

Testing is a ubiquitous skill for modern developers. The consensus, for the most part, is that testing is beneficial for software development. There are fads and buzzwords that come and go, but testing is timeless. I want this book to help the reader keep their testing skills fresh throughout their career.

If you're worried that you know about testing but you're not using it enough to benefit from it - this book is for you. Tests are more than a requirement. They are not a chore. I want to show you how to use tests like you use all of your other go-to development tools. When testing becomes a tool that you can't live without, you will not have to worry about this skill deteriorating. I'm hoping that you may use this book to develop testing into a tool that you use to benefit both yourself and your code.

How this Book is Organized

This book is organized into two main sections:

- Discussion of Legacy Testing Problems and Approaches

- Walkthrough of Testing a Legacy Application

Chapter 1: Legacy Testing Problems and Approaches

- Testing Legacy Projects Requires a Different Approach
    - Your task with the legacy project will likely be one of four change types. There are risks that you will be taking by introducing any of these changes to a legacy codebase. Testing is a risk mitigation strategy for these changes.
- Asking More Questions of a Legacy Project
    - It's ineffective to simply try to cover a legacy project with tests. We need to ask different questions than simply "What should I test." We need to be strategic with legacy testing.
- Seams and Hard-To-Test Code
    - The hardest thing about testing legacy code is finding the right places to isolate functionality and exercise it with tests. We need to listen to tests and make good refactorings based on what they tell us.
- Using Tests to Gain Productivity
    - The debugger isn't the best place for rapid development with a legacy project. Testing is faster. Testing allows us to maximize our productivity by writing code that stays with the project and leaves breadcrumbs as you dive into the legacy app.

Chapter 2: Introducing the Walkthrough

- The Scenario

    - We are simulating a unique legacy testing scenario that, from the surface, is not conducive writing tests. Once we begin, however, we will reap the benefits of using tests to take command of the legacy app.
- The Project
    - We have a 10-day deadline to add a feature to a legacy project with very little information about how it's used and how it was built.

Chapter 3 thru 9: Adding a Feature to a Legacy App using Tests

- The project will be broken down into a timeline to gain an understanding of our productivity losses/gains. We use tests to learn the codebase and implement the feature.

Chapter 10: Conclusion and Final Thoughts

Testing Legacy Projects Requires a Different Approach

To build on why there is such a difference between learning testing and actually testing a legacy project, let's talk about how comfortable you are making change.

In Michael Feather's book “Working Effectively with Legacy Code” he outlines “four reasons to change software”:

- Adding a feature
- Fixing a bug
- Improving the design
- Optimizing resource usage[2]

These different types of changes spawn different ideas of how much change and risk you are introducing to the existing code. The amount of risk you're assuming should guide your testing strategy. I know I said I wasn't going to talk about why we test, but this reason should be clear. You are testing to reduce risk.

What kinds of risks are you taking when you start working with a legacy codebase?

- Risk of breaking some existing functionality
- Risk of your new code not doing what it's supposed to
- Risk of your stakeholders rejecting what you did
- Risk of the behavior of the application changing

---

[2] Feathers, Michael C. Working Effectively with Legacy Code. Upper Saddle River, N.J: Prentice Hall PTR, 2004. Print

- Risk of losing trust with your users
- Risk of being reprimanded by your boss
- Risk of downtime costing the company money
- Risk of losing confidence in yourself as a developer
- Risk not being able to come up with excuses for your code not working

All of these risks have something in common. They cost money. Some costs are more obvious than others, e.g downtime keeps users from being able to do their jobs. If a salesperson can't sell a product to a customer for a day, that's opportunity cost. If you lose trust with your users, that's opportunity cost. If your stakeholders reject your feature, that's your time and salary sunken.

If you think of your time as costing money, the not-so-obvious costs become clear. Losing confidence in yourself as a developer can make you cautious and afraid to try new things. Having to constantly come up with excuses for why your code isn't working is a huge drain on your internal resources. Being reprimanded by your boss causes you stress. It forces you to waste time figuring out how to deal with that stress. These things take you away from focusing on what is essential to both your career and the people that depend on you. These things cost time and time is money.

So what does this have to do with testing? You're more likely to be working with a legacy project than not. Therefore, you are more likely to be doing one of the above four changes. You need a toolset for reducing risk. It's not just about being able to write tests, or even being able to write good tests. It's about your current situation and what is essential for your success at this point in your project.

For example, you will probably have to sacrifice writing a terrible test just to get started. By terrible, I mean you may have to smell up the test code - too many mocks, multiple assertions, copy/pasted test fixtures, even copy/pasted implementations of

SUT code! All of these may happen, but in the end **you will have started**. You'll be shipping. You'll be learning about the code. Most importantly, you'll be **writing** code instead of reading it. This has an immense effect on your productivity.

My favorite analogy is to a rock climber. A rock climber examines his route up the side of a cliff before he starts his ascent. He can't anticipate every spot he's going to place his feet, every crevasse he's going to grip, every loose pebble he's going to encounter. He also can't spend all day at the bottom of the cliff. He needs to make progress so that he can get to 50 ft and get a better idea of his next move. There will always be a chance that he will encounter something unexpected. He will not know everything until he gets there and puts his hand on the rock. He will use ropes and anchors to establish intermediate points of protection on the way up. After setting an anchor, he'll make small adjustments to his route. Worst case, he's safe to fall back to his last anchor if he makes a mistake. This is a much better approach than planning the entire route and expecting his plan to work out perfectly.

This is the same approach you can take to testing. Nothing will go perfectly. As David Heinemeier Hansson says in his blog post "Software has bugs. This is normal.": "The only reliable, widely used way to ensure impeccable software quality is to write less software that does less stuff."[3] If you're like me, you try to do more planning in order to lower risk. You and I, like the rock climber, aren't clairvoyant. We can read code, but until we actually touch it we can't know everything that will happen when we touch it.

---

[3] Hansson, David H. "Software Has Bugs. This Is Normal. - Signal v. Noise." Medium. N.p., 2016. Web. 12 June 2016. https://m.signalvnoise.com/software-has-bugs-this-is-normal-f64761a262ca#.78z2skaf3

Our approach to legacy testing is going to be different. It requires a different approach that draws from the various dogmatic testing strategies out there. Our strategies will focus on lowering risk, lowering cost, and shipping code.

Legacy testing can't be treated the same as greenfield testing. We are usually faced with a unique scenario that we have never seen before. We don't know everything about the legacy app, especially when we first open it in the editor. We are taking risks because we don't know everything about the app. The biggest risk is making change to the code and not knowing our effect on the existing behavior. If we don't mitigate risk, it costs money and time.

Tests are used to mitigate risk, both in greenfield and legacy development. With legacy code, we can use less conventional testing strategies as long as we continue to mitigate risk. Instead of considering testing a requirement, we should consider it a tool - just like your editor, your favorite patterns, or the debugger.

Asking More Questions of a Legacy Project

Let's dig deeper into why legacy projects require a different approach to a test strategy. As Feathers describes in “Working Effectively with Legacy Code” we have to consider how much behavior we are impacting with our change. All of the four types of change: feature, bug, refactoring, optimization will affect some behavior of the application. In addition to the behavior that the change will directly address, there is existing behavior that we need to preserve. This is the unknown. This is the risk that we want to mitigate.

In an ideal world, we would cover every code path with tests to achieve 100% coverage across the entire system. Maybe our boss realizes this is not possible and declares that we need 50% code coverage. The trouble is, we can't lay a blanket over all of the existing behavior - we don’t have time.

Same is true for bullet-proof vests. An police officer equipped in a complete bomb-squad suit loses mobility. While he would be fully protected, he has other jobs to do. He must be able to get in and out of his car, talk to victims, and run at a reasonable speed. A bullet-proof vest is lighter weight and protects his vital organs. The vital organs have the highest risk of fatality from a gunshot. In our legacy project, the existing behavior that will be affected by our change has the highest risk.

What do I Test?

I've asked myself "What should I test?" many times. In a greenfield application, you likely know what bugs are lying underneath a block of code that **you** wrote. If you

practice TDD, then what to test is clear to you since you are driving your implementation with tests. With legacy code it's ineffective to randomly pick code out and test it. Depending on the size of the project, you might start out strong, testing each class one-by-one. Like a sprint, however, you will eventually feel like you're not getting anywhere. That's why I suggest writing tests in parallel with any feature, bug, refactoring, or optimization that you're tasked with.

You're likely tasked with something to do with the production codebase that's not test-related. If that's the case, I recommend focusing on writing tests that benefit your efforts to deliver on whatever you're tasked with. One major example of this is a feature. It's likely that a feature isn't going to be developed in isolation from the existing legacy code. The legacy code affected by the new feature is a prime candidate for writing tests. It not only protects existing functionality, but it makes you familiar with that code and how you're going to integrate your new code with it. There are more benefits, and we'll see them when we discover a good slice of existing legacy code that we want to test in our sample project.

What am I Doing?

Yes this is an obvious question. Probably one that you know the answer to. "I'm fixing a bug with this legacy app." There's more depth, however. What are you doing in reference to your role as a resource for your employer/client? You may be fixing a bug, but is the bug impacting business continuity? Is it a bug that is stopping the business from gaining revenue? Who is asking for the feature? Is it a habitual "feature-requester?" Who's telling you to refactor that code - is it your manager, or another developer who just read a blog post and says "you have to use X pattern, it's so much better than what you're doing…"

It's important to know what impact you're making on the business with your change. If it's an impact to something that gives your business its competitive advantage, then testing should play a bigger role for you in the project. You'll want a high percentage of your time to be focused on writing good, maintainable tests. If the change is to something that doesn't give the business its competitive advantage, this is a chance for you to try new things with your tests, e.g. new mocking frameworks.

Your situation should play a role in your testing strategy. You don't want your testing efforts to be seen as a waste of time. You need to always have an idea of the ultimate value of your tests. This is another reason that having a test coverage metric isn't as effective as testing to facilitate a change.

With legacy testing, a test coverage metric like 50% isn't going to give you the most value out of writing tests. If you were aiming for a test coverage metric, you might ask: "What do I test?" With legacy testing, we should ask questions like:

- "How much will this change affect the existing behavior?"
- "Where am I going to integrate my new code with the existing code?"
- "How business-impacting is this bug?"
- "How often is this code executed in production?"
- "Will my teammates be able to debug this if I'm not around?"
- "What's at stake with making this change?"

It's important to stay focused on what's essential when trying to test a legacy app. You can end up wasting your time building testability into the app when it's not essential. It will be harder to create testability where there is none, so make sure you prioritize your testing efforts to maximize their value.

Seams and Hard-To-Test Code

There are a few reasons it is hard to measure the amount of existing behavior that is at risk when we are making a change. A prominent reason has to do with highly-coupled dependencies. In a legacy project, you will find that code with high-coupling is hard to test. It will be difficult to find where to begin infiltrating the code to get to the actual "unit" you want to exercise with your tests. Consequently, it will be difficult to simply pick a spot and start testing.

Feathers defines a “seam” as “a place where you can alter behavior in your program without editing in that place"[4]. When I first began testing legacy code, both MVC and non-MVC, it was difficult for me to find seams. My predecessors were unaware of the benefits of testing. They were also unaware of refactoring principles and how to identify code smells as described in Refactoring by Martin Fowler[5]. The result: long methods with too many responsibilities.

One particular responsibility in these long methods made them hard to test. This was the `new` keyword. Multiple `new` keywords in these long methods were a clear sign of coupling for me.

So if you are asking the question: "What do I test in this project?" you may have to ask: "What can I test?" instead. Knowing where you're going to have to pry apart

---

4 Feathers, Michael C. Working Effectively with Legacy Code. Upper Saddle River, N.J: Prentice Hall PTR, 2004. Print

5 Fowler, Martin, and Kent Beck. Refactoring : Improving the Design of Existing Code. Reading, MA: Addison-Wesley, 1999. Print.

seams will affect your test strategy moving forward, regardless of what items you have identified as "test-ready".

The seams discussion leads to a more general reason we have to adapt a test strategy for the legacy project - Hard-To-Test Code. Again, you might decide with authority: "I need to test that this controller action method returns this ActionResult." If the action method is 100 lines and `new`s up four dependencies, the method is Hard-To-Test. This is a test code smell from Gerard Mescaroz's book "xUnit Test Patterns: Refactoring Test Code"[6]. When smells originate from the System Under Test (SUT), Hard-To-Test code causes a lot of difficulty writing tests.

[6] Meszaros, Gerard. XUnit Test Patterns: Refactoring Test Code. Upper Saddle River, NJ: Addison-Wesley, 2007. Print.

Using Tests for Productivity

Remember Feathers' four primary reasons to change software:

- Adding a feature
- Fixing a bug
- Improving the design
- Optimizing resource usage

Without testing, how quickly can you get up and running with one of these changes to a legacy app? Looking back at the way your stakeholders give you tasks to complete, how easy is it going to be to kick that feature out?... fix that bug?... clean up code?... make it faster?... At a minimum, the only information you have is the code and maybe a UI. Without tests, you'll inevitably be immersed in debug mode. Actually, your world will probably look like this:

- Open Visual Studio
- Open up the email with the project requirements
- Set breakpoint where you might find something
- Hit Debug
- Wait
- Go to http://localhost/SomeRoute/
- Fill out a form
- Click Submit
- Wait

- Hit breakpoint
- Examine locals, hover mouse over variables, run something in immediate window, etc
- Make a mental note of something learned
- Write some production code
- **Repeat...**

There's also the possibility that you set your breakpoint too late. You are then forced to repeat before you are able to gather any information. That repeat step is what I hate most about the debugger. At a breakpoint, you are given a single time span to gather information about the state of your app. You are freezing the app in a state that you assume the user will be in. This is definitely useful, until your human mind gets in the way.

Have you ever set a breakpoint after you needed to? Have you ever left a breakpoint from a previous debugging session, forcing your app to stop where it didn't need to? Have you ever forgotten to set a breakpoint and debugged anyway just to blow past any useful state information you may have gathered? I have... and it was slow, painful, annoying, unproductive and left me in search of a better way to fully understand legacy code.

Shipping

Again, I don't want to tell you why automated testing is good for the health of the application. This area has been explored and many experts have their own take on why we test and how to get the most out of tests. What I'd like to focus on is how you can

apply a testing strategy to a legacy application. A huge reason to keep reading is that when you write tests, you are shipping.

How many lines of code did we write while we were repeating debugging sessions over and over again? Maybe you're a "rockstar" and it only takes a single debug session to add a feature to an unfamiliar codebase. Good for you. I'm not a rockstar. I'm a beginner automated tester and I don't know where to start with this legacy app. I have deadlines, I want to get better, and I need to get started.

I want to show you how testing lets me actually ship code early. Instead of the above basic debugging scenario, let's learn about the app by writing some tests:

- Open Visual Studio
- Open up the email with the project requirements
- Find somewhere in the code that is related to your requirements
    - A sibling feature
    - A parent feature
    - An API call
    - A Database call
    - Existing infrastructure code that you'll need
    - A confusing method
    - Copy/paste-able code
- Find a seam (if possible)
- Write a test
- Make the test build
- Make the test pass
    - Ugly or pretty, whatever it takes
- Discover results

    - Send property values to stdout
    - Exceptions thrown
    - Build errors
- Assert something
- Refactor the test a bit
    - Rename it clearly
- Keep moving **forward**

The first time we write a test may take time. We will need to get our test project set up, add a test framework, and make sure our test runner recognizes the test framework. Once we have this "walking skeleton"[7] in place, our ability to write tests and exercise production code improves drastically.

It's this moving forward that is where we gain the most from writing tests early. It allows us to ship something early. Shipping early makes you a better developer. It keeps your hands on the keyboard. You're as close as you can get to writing production code, when you're writing code in a test project.

What sounds better?

- debug...
- breakpoint...
- continue...
- what was that property value?...
- debug...
- breakpoint...

---

[7] Freeman, Steve, and Nat Pryce. Growing object-oriented software, guided by tests. Upper Saddle River, NJ: Addison Wesley, 2010. Print.

- write something

or

- write test...
- learn something…
- write assert...
- write rename/refactor...
- write test...
- learn something...
- write assert...
- write rename/refactor...
- write test…

It's always better to write. Our approach in our sample project is going to be focused on shipping test code quickly so that we can ship production code quickly. To further illustrate this, I'll break our progress down into a time frame to see if we can keep the number of days lower than it would normally be. We'll use tests to squeeze out productivity all the way up to our deadline.

Testing isn't usually sold to potential practitioners as a way to increase their productivity. It's a benefit that I only realized once I started testing legacy code. Writing a test that stays in the test suite or in source control is more permanent than debugger information. Having a test written for a part of my coding progress is a breadcrumb for me to remember exactly what I was trying to accomplish at that time. It's also faster to not have to spin up the debugger every time I want some state information.

Refactoring your code to the point where you can get state information through tests is a good approach. Having clear outputs from a function is one reason why func-

tional programming is said to be more robust and bug-resistant. If you can refactor your code to the point where you can test it and find out exactly what it was meant to do, you likely have a good design.

Writing code is always going to be more productive than reading it. While you are writing code, you are learning what's going on. You are also unearthing very precise refactoring to make to the production code. You learn the history of the development of the app - testing a branch of logic can tell you why it's there. The debugger can give you this information too, but it will always cost you time if you rely on it too much.