A(n) (Re-)Introduction to Test-Driven Development

Even if you feel comfortable with the basics of test-driven development, I strongly believe in building a ridiculously strong foundation of basic skills. For this reason, I've gone right back to the beginning. Please don't skip this material, nor this exercise. You'll almost certainly learn something by doing it.

If you've never attempted test-driven development before, then this exercise will introduce you to the very simplest version of the technique. No matter how complicated or sophisticated your understanding of TDD, of design becomes, you can *always* return to the fundamentals you learn here.

Overview

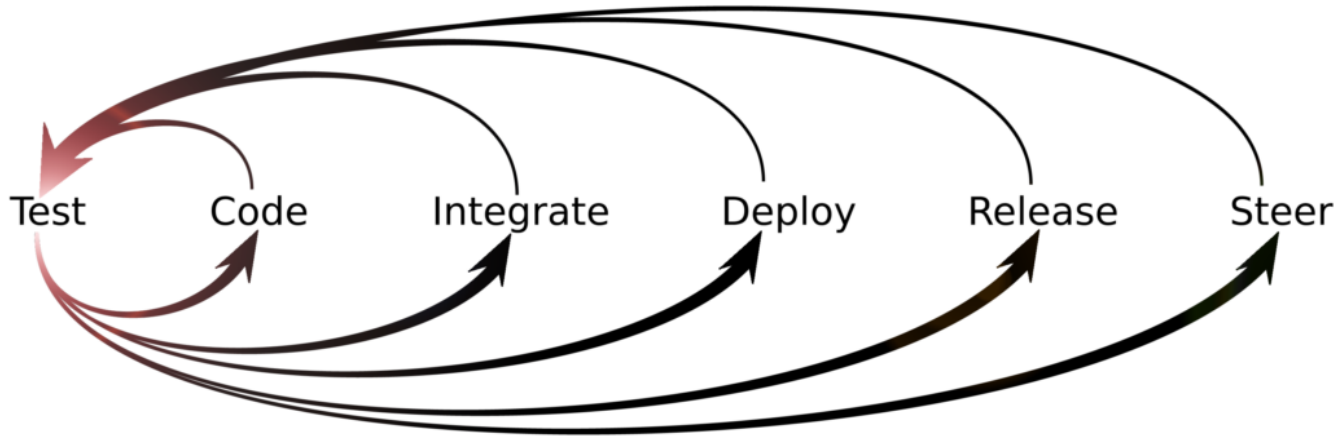
Do the following.

1. Read this entire article.
2. Do the exercise.
3. Read as many of the references from this article as you have energy to read.
4. Ask your questions in the mailing list.

Back to the Beginning

If you've never read Ward Cunningham's wiki, then consider yourself infected. Visit http://link.jbrains.ca/tdd-at-c2 and you'll see how the then-new XP (Extreme Programming) community thought of TDD.

(Read the rest of this article first, then go back and read more from Ward's wiki.)



TEST-DRIVE ALL THE THINGS! Source: http://www.zeroplayer.com/tdd.png

This diagram shows how what we now consider TDD fits into the overall cycle of shipping features. You'll notice that this diagram doesn't show **design** as a separate activity; instead, **code** includes **design**. Indeed, this is a key point of TDD: we design as we write tests, we design as we write production code.

I don't want to argue where to put "the boundary" of TDD in this diagram, so please don't ask. Instead, I encourage you to start by driving your code and design decisions with tests, then as you become more comfortable doing that, try driving other decisions with tests.

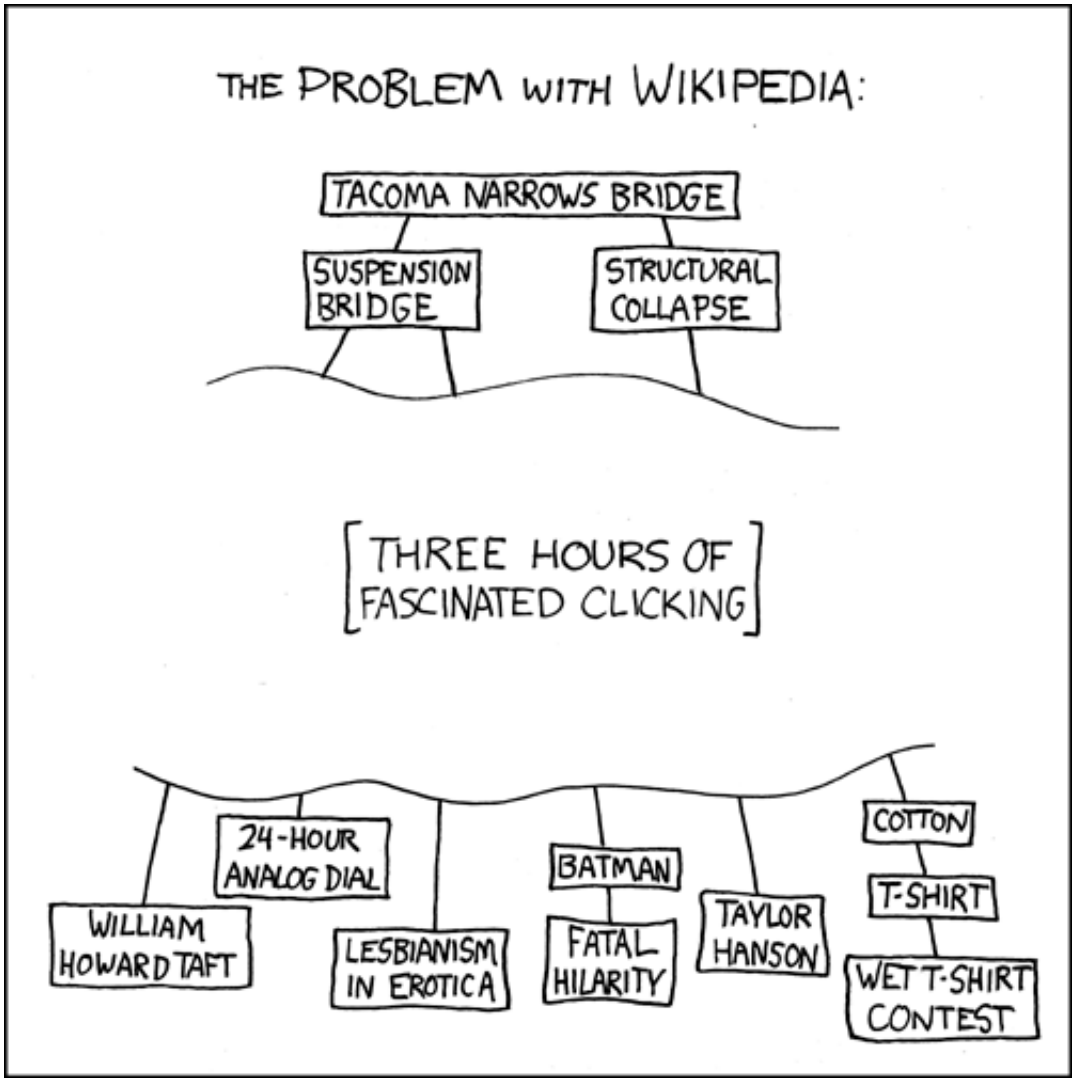
The World's Simplest Instructions for TDD

This comes straight from the Ward's wiki page on TDD.

> *When you code, alternate these activities:*
>
> 1. add a test, get it to fail, and write code to pass the test (DoSimpleThings, CodeUnitTestFirst)
> 2. remove duplication (OnceAndOnlyOnce, DontRepeatYourself, ThreeStrikesAndYouAutomate)

When you read this at Ward's wiki, of course, those camel-case words become links to other articles. Remember: read the rest of this article first, then go back and read Ward's wiki.

Seriously. Don't jump to the article at Ward's wiki yet. If you try to read Ward's wiki now, you'll spend at least 6 hours clicking link after link.



Ward's Wiki had this problem way before Wikipedia. Source: http://xkcd.com/214/

That's it. Already, with such simple rules, we establish a bunch of useful guidelines.

- Write no production code without a failing test.
- Watch a new test fail before you try to make it pass.
- Write easy code to make new tests pass.
- Whenever all the tests pass, improve the design by removing duplication.

What These Instructions Give You

You might already notice some benefits from following these instructions. I tried writing these benefits in a particular order, but I found so many, that I simply decided to dump them here for you. Perhaps they fit better in a cause-effect diagram.

- You'll tend not to write untested code.

- You'll tend to work in smaller steps.
- You'll tend to feel more confident about the micro-steps you've completed.
- You'll tend to spend most of your time with a working, but incomplete, code base.
- You'll avoid scattering duplicate code throughout the code base.
- You'll tend to write small batches of code at a time.
- You'll tend to write less code than you might otherwise write.
- You'll tend to know more precisely what code you need to write.
- You'll tend to notice earlier when you make a mistake.
- You'll avoid letting mistakes compound each other.
- You'll avoid needing to make the same change in multiple places.
- You'll tend to change code more confidently.
- You'll tend to change code more aggressively.

I've written these as *tendencies*, because of course we don't practise TDD perfectly strictly. Even after you've developed absurdly high discipline, you'll have always developed trustworthy judgment that will allow you to break the rules sensibly. For now, though, I strongly suggest that you err on the side of more diligence, more strict practise, more following the rules. *You already know how to design your way; you've come here to learn how to design a different way.*

Of course, these benefits lead to yet more benefits over time, but I don't want to overhype the topic. I have no question that TDD has led to transforming my life in ways that I never imagined, but if it only ever helps you become a more disciplined, accomplished software designer, then I think we can agree that it will have done its job, no?

Enough Talk; Let's Code

Before I describe the actual exercise, I want you to make sure that you have set up a development environment that maximises safety, which improves confidence and aggressiveness. This relates to the XP value of "courage", which the community originally called "aggressiveness" (in the best possible sense).

The rules of TDD encourage us to take small steps, which we can use to make *undo* just as safe and easy as *do*. If you've seen two-year-old commented-out code in your code base, then you've seen how people write code when they feel fear. I want you to fear only the things you *need* to fear, like death by lion; I don't want you to fear code. I suggest these ingredients for a confidence-building, courage-friendly development environment.

Your Tools

- a version control system with an off-site backup
- a testing framework or library for your programming language
- a text editor or IDE that you want to master
- a mechanism for running tests with (at most) a single keypress while still in your editor
- pen, paper, index cards

For example, when I work in Java, I use these tools:

- IDEA 12 (I can run tests with a single keystroke)
- JUnit 4

- git + either github (public projects) or ProjectLocker (private projects)
- a pen, a stack of scrap paper, at least 100 index cards

When I work in Ruby, I use these tools:

- vim
- RSpec
- guard to monitor the file system and run tests when files change
- git + either github (public projects) or ProjectLocker (private projects)
- a pen, a stack of scrap paper, at least 100 index cards

> Before reading any further, set up your working environment. If you believe that you already have it, then try this: set up a new, empty project, write a single failing test, run and watch the test fail with at most a single keypress, commit the failing test to version control, and publish your single failing test off site. If you can quickly and easily check out and run your failing test on another part of your file system (or another computer!), then you pass the test. Once you can do this, feel free to read on.

Exercise: Fraction

I want to start with a simple exercise that you can implement in a single module (class, if you plan to use classes), so that you can focus on very diligently following the rules of TDD. Once again: **even if you think you have mastered the basics of TDD, I want you to practise them**. If nothing else, then you can start practising patience, which every good evolutionary designer needs to practise!

You know fractions. If you don't know fractions, then read the Wikipedia page on the topic. Smalltalk ships with a library for doing exact arithmetic with fractions, but other languages do not. You will start building a library to do basic, exact arithmetic with fractions.

Really Quick BDD

We're doing behavior-driven development (BDD) right now, so enjoy it. If you expect Cucumber tests, then prepare yourself for some disappointment.

I'd like to focus on basic arithmetic: add, subtract, multiply, divide. We might never need more than that. I feel pretty confident that we can build the rest on top of these features.

I need fractions to remain compatible with whole numbers, so for example (the magic word of BDD), 4+9 needs to work the same as fractions as it does for integers. (That sounds like a test, so you should probably write that down.)

I'd like eventually to combine fractions with built-in numbers, but I can wait for that. Your language might let you do some syntax magic to make it easy to add built-in integer 5 to fraction 4/3. If so, then don't let that magic captivate you to the point where it stops you from getting fraction 5/1 + fraction 4/3 working.

I think we can agree that adding a floating-point value to a fraction doesn't make much sense. I don't need you to do anything clever here. If you have a type checker, and it stops me from trying to subtract float 2.45 from fraction 18/5, that works for me.

Oh yes... two more constraints:

- Always express fractions in *lowest terms*. For example (another test!), fraction 4/6 magically becomes 2/3.
- Always express fractions as *improper* and not *mixed*. For example, I want 7/5, not 1 2/5.

I can't think of anything else important, and this feels like enough to get started.

What We Just Did

We just did some BDD, you and I. We following some useful guidelines:

- get stuff out of your head quickly
- when in doubt, provide a concrete example to illustrate what you mean
- state clearly what you want, what you don't need, and what you don't want
- stop when you've said enough to get started

Keep these guidelines in mind. They will also help you write tests, write code, and design.

Do This Now!

Now for your exercise.

At a minimum, ship a module that can add fractions exactly. If you want to go further, then ship a module that can perform all four basic arithmetic operations exactly on fractions: add, subtract, multiply, divide.

Here "ship" means that **I can download your module and use it**, so whatever packaging mechanism your language provides (JAR, gem, whatever), ship one of those things. **I know! Start by writing one simple test that uses your new, awesome library *as though you've already shipped it*!** Trust me: one of these tests for your add function/operation/whatever will suffice, so don't go crazy with details at this high level.

When you write this test, you'll already have to make a bunch of decisions and answer some questions:

- how will I package my module?
- what will I name the module?
- what will I name the function/operation?
- how will I automate packaging my module?
- how will I publish my module for the world to use?
- *when* should I publish my module for the world to use? (Hint: probably not on every micro-commit of your code.)

Now you see why I recommend writing only one such test. With the first test, you already have to decide a lot. In your situation, I'd probably write a test that adds 7/3 and 4/5 using your (eventually-)published API, then stop there. (The expected result, by the way: 47/15.)

After you've written this test, run it, and seen it fail, set it aside and get down to the details of how to add fractions. In other words, make a Test List.

Making a Test List

Grab your pen and paper. (I meant it when I included them in your tools.)

Write a single example of adding fractions. It will amaze you how many people don't do this correctly the first time. They can add fractions, but they forget to write a complete example. Either that, or they overthink it. A complete example of adding fractions looks like this:

a + b = c

Choose your own fractions for a and b, then compute c by hand. (I've intentionally not put real values here, because I don't want to anchor you to any specific examples.)

Your list of tests, then, can look like a bunch of expressions of the form a + b = c. You might want to add a word or two next to an example to remind you what makes the example important. You might prefer to jot down the words first, then circle back and write a complete example next to those words. Follow one of the guidelines I described earlier:

> *Get stuff out of your head quickly*

I write a Test List for this purpose: to get stuff out of my head quickly, which lets me then focus on making a single test pass at a time. *I don't want ideas for future tests to distract me!* Neither should you. Don't rely on your memory, because either you'll forget, or you'll distract yourself by holding too many things in your head at once. ("It's no good to anyone else in your head, Mozart; write it down.")

I also write a Test List in order to think through enough examples (tests) to get started. So do that now:

1. Write an example on your test list. Either write a + b = c or a few words to remind you what the example means.
2. Repeat until you can't think of another useful example. If about 15 seconds go by without another example popping into your head, you've probably done enough.
3. Look through your list, and if anything seems unclear, clarify it. This might lead more examples to pop into your head, in which case, write them down until the 15-second rule tells you to stop.

After this step, you have either a page or some index cards with a bunch of examples (tests). Now you need to decide where to start. In your position, I'd start with the simplest test you can think of. Remember all the decisions you'll have to make to actually write, run and watch fail the first test.

- what shall I name this module/class?
- what shall I name this function/operation?
- what shall I name this package/namespace?
- how shall I export this code for public use?
- how shall I pass parameters to this function/operation?
- will my choices clash with existing libraries?

- how shall I instantiate any objects that I might need?

With so much to consider, I always choose the simplest example I can find to write first. I didn't want to anchor you before, but I don't feel bad about doing that now. I choose this example:

> *0 + 0 = 0*

You did write this example, didn't you?

> Don't feel bad. We've all worked on projects where we never took any time to think about what our software should do with, for example, an empty database. The first customer to incorrectly install our software got a stack trace, it embarrassed us, we blamed the customer... now we don't do that anymore. We consider these "empty" cases, even if they don't "make sense" to us. We practise TDD precisely to encourage us to consider these cases and deal with them *before* we create embarrassing messes.

Now you follow the rules of TDD, to which I've added some details related to our Test List:

1. Choose the simplest test from the Test List.
2. Write the test, run it, watch it fail.
3. Write easy code to make the new test pass, while keeping all the old tests passing.
4. Cross off the test from the Test List that now passes.
5. Remove duplication, keeping all the tests passing.
6. Repeat until you've crossed everything off the Test List.

At any time, if another test pops into your head, then *get it out of your head quickly*. Write it down on the Test List, then forget about it and refocus on your current micro-step.

Ahem... Version Control?!

Oops! I forgot about version control. I always follow this simple guideline:

> *Any time the tests pass 100%, I may commit.*

For you, right now, I recommend committing every time a new test passes and after every change that removes duplication. I also recommend this stronger guideline:

> *Any time the tests pass 100%, if you feel the slightest urge to commit, or the slightest guilt for not committing, then **you must commit**.*

If you prefer, use this more refined set of instructions:

1. Choose the simplest test from the Test List.
2. Write the test, run it, watch it fail.
3. Write easy code to make the new test pass, while keeping all the old tests passing.
4. Commit your changes with a message like "module X now does new thing Y" or "module X now handles case Y".
5. Cross off the test from the Test List that now passes.
6. Remove duplication, keeping all the tests passing.

7. Commit your changes with a message like “removed duplicate X from function Y”.
8. Repeat until you’ve crossed everything off the Test List.

I can’t think of anything more that you need to do the exercise. You’ve completed the exercise when you (or someone else) can download your packaged API and (at least) add fractions with it.

Have fun!

Quick Summary

1. Focus on following the steps strictly and diligently. Pay attention to what you’re doing. Make note of any “Aha!” moments, so that we can discuss them.
2. Set up your environment with care and attention. You don’t just write code: you run tests, you commit to version control, you package and publish your API.
3. Lean on your Test List and the guideline to get stuff out of your head quickly, so that you can focus on your current micro-step.

One More Little Thing

If you have trouble with time “getting away from you” or spending too much time going down “the wrong path”, then try the Pomodoro Technique. For your purposes, you just need a timer, since your Test List acts as your to-do list.

1. Set the timer for 25 minutes.
2. Start the timer.
3. Start working.
4. When the timer buzzes, stop working. If you’re worried about forgetting something, then get it out of your head quickly.
5. Set the timer for 5 minutes.
6. Take a break. Get up; walk away; take the timer with you if you plan to go far.
7. When the timer buzzes, get back to work.
8. Repeat as much as you like.

This felt really stupid the first 10-20 times I tried it. It got better. I noticed that I completed micro-tasks more quickly. I also noticed that if I got nothing done in 25 minutes, then I didn’t really understand my current task, so I took some time to think about it, breaking it into smaller micro-tasks, then complete them more quickly. I might go down “the wrong path” for 15 minutes instead of 2 hours, it felt like an improvement.

More importantly, now I can “get into flow” in just a few minutes, and it amazes me what I can get done in 25 minutes now.

References

C2.com Wiki Community, “Test-Driven Development”. http://c2.com/cgi/wiki?TestDrivenDevelopment

James Shore and Shane Warden, "The Art of Agile Development: Refactoring". James excerpts their book of the same title, in which they describe their approach to refactoring, including the notion of Shotgun Surgery, a symptom of letting duplication live in your code base.

Andy Hunt and Dave Thomas, The Pragmatic Programmer. This book ranks in the top five of influencing my practice as a programmer. I found it an easy read, because it consists of several short, relatively independent articles with highly effective tricks that I could apply minutes after learning them. I remember quite clearly their advice to "learn one editor and learn it well". In 2001, I chose Eclipse. (You young programmers don't know, but Eclipse used not to suck.) In 2012, I chose vim.

Liz Keogh, "Step Away From the Tools". Most of us, when we learn a new technique, want tools to help us get started. These tools can absolutely help us get started, but if we don't step away from the tools as we learn, then we start to believe that the tool *is* the technique. When we do this, we miss the point. I wish I'd written this article; it matches almost perfectly how I feel about the issue.

J. B. Rainsberger, "Primitive Obsession Obsession". I wrote this article to discuss the code smell Primitive Obsession. In the context of designing your fraction module, I more strongly recommend the references in that article over the article itself.

Kent Beck, Test-Driven Development: By Example. This book remains a classic introduction to the topic. Kent wrote the main example in Java, but the last third of the book shows test-driving a testing library in Python *using itself*, which I think everyone ought to try at least once.

http://www.pomodorotechnique.com. I think some people have taken the technique way too far, but I continue to benefit from the basic technique in my daily work. Don't let it become a religion, and you'll do just fine. Remember the goal: completing valuable micro-tasks. Whatever it takes to increase your ability to achieve this goal, do it.