# Test Driven Development – An Overview

## About Me

* Professional Developer since 2000
* Practicing TDD for roughly 8 years
* Co-founded Codera Ltd this year
* Working with The Test People on Client site

## A Bit Of History

* Not based on new ideas
* To shorten the total development time, some formal test documents (such as for [acceptance testing](http://en.wikipedia.org/wiki/Acceptance_testing)) have been developed in parallel (or shortly before) the software is ready for testing. A NASA independent test group can write the test procedures, based on formal requirements and logical limits, before the software has been written and integrated with the hardware.
* One of the earliest agile methodologies that came out was called eXtreme Programming and was a very development centric approach.
* XP came out of the work Kent Beck did at Chrysler around 1996-99. His book eXtreme Programming Explained was published in 99.
* Under the rules of XP the coding part describes TDD without giving it a name.
* In 2003 Beck published a book entirely dedicated to TDD. Here he stated his belief that he merely “rediscovered” TDD.
* This year a blog post from DHH (David Heinemeier Hansson) the creator of the Ruby On Rails framework denounced TDD and claimed as a methodology it was dead.
* This “pronouncement” led to a lot of online discussion on twitter & blogs with lots of prominent developers adding their two cents worth.
* Led to a series of hangouts and discussions between DHH, Kent Beck and Martin Fowler, which are available on the web.

## What is TDD?

* A development technique
* It’s a way to drive the design of the code as we go.
* Quickly write a failing test
* Get the test to pass as quickly as possible. Commit any sin necessary to get the bar green as fast as possible. Copy and past code, hard code the answer, etc
* Remove duplication. Often developers will talk of tidying up the code or “refactoring”.
* It is not when the tests are written shortly after the code

## Why Develop Like This?

* The goal of Test Driven Development is to provide clean code that works.
* One of the main benefits of using TDD is that you are dealing in concrete examples making it quicker and easier to crystalize and communicate what you are trying to achieve. This also makes it much easier when talking to other developers or testers because you can talk in terms of the expected behavior of the system. “So, if I call the calculation passing these two values then I expect the tax rate to come out at Y”, etc.
* As you apply TDD you find that you get into a rhythm, writing tests, getting them to pass, tidying up the code and then moving onto the next test. The important thing is to take small steps and keep moving towards your goal.
* Initially it feels very unnatural to people to take such small steps. It’s quite common to think the solution to a problem is obvious and implement it in one go only to find that it’s not as straightforward as initially thought or there is some complicated corner case to contend with.
* One of the most important things to achieve this steady rhythm is a test suite that can be run and provide immediate feedback. You don’t want people sat around waiting for validation and then losing focus or their train of thought.
* Fear – the overall effect of TDD is to manage fear and allow you to change the code with confidence.

## Why Doesn’t Everyone Do It Then?

* The cost of maintaining tests.
* This is two-fold.
* Firstly there is the time taken to run them, i.e. if you have a suite that takes minutes or hours then you have people sat around waiting for feedback.
* Secondly there is the cost of badly written tests that are prone to breaking, reducing confidence in the test suite and ultimately leading to the kind of code rot we are trying to avoid.
* Some people don’t like it because they argue there is no up front design and it’s an excuse to not think about what you are doing. There is nothing in TDD that says you shouldn’t think about the general design before you start.
* The flip side is other people say it leads to overly complex design. This is one of the big things that DHH accused TDD of. He argued that people were overly obsessed with getting quick running tests at the cost of compromising the system design by introducing layers and layers of indirection and mocking.
* Part of his issues arose because Rails doesn’t lend itself to decoupling objects from the database layer and people went to great lengths to work around the framework.
* Quite often you will come across teams or individuals who will say they tried TDD and it didn’t work so they abandoned it.
* Most of the problems described here can be explained by the fact that there is a steep learning curve when it comes to TDD. The rules are incredibly simple, write a test, make it pass, refactor, repeat. However there is great skill involved at each stage. What level do you write your tests at? When do you mock something out versus when to use a real collaborator? How big a jump can you take getting this piece of functionality working? It’s only after working in a TDD manner for a number of years and living with the test suites and the codebase do you find the things that work over time and the things that hold you back from changing the code base.

## The Secret Sauce

* Refactoring is what makes TDD really work. I think it’s quite common for people to think that it’s the implementing a single test at a time that is the most important. That is what helps you keep the problem at a manageable level in your head. What helps over the long term is refactoring.
* In TDD By Example Beck described the process as test, implement and then remove duplication. Generally we talk about this last step as refactoring which covers more than just removing duplication, however the primary focus Beck talked about was removing duplication,
* Duplication is what makes code hard to change over time. If we ever need to make a change to the software in the future and there are 4 variations on a calculation then there are 4 places we need to change the code. If we had refactored the code as we went then we would have extracted the common part of the calculation and there would only be one place to make the change.
* Use refactoring to hide details behind abstractions. This works in both tests and implementation code. For instance if you are testing a GUI which has a form the user needs to fill in hide it behind a submit form method. The act of filling in the GUI is taken care of by a specific implementation.
* Now when the business come along and say they want to expose the same functionality to other clients but via a web service you can provide a new implementation that fills in the request and sends it to the web service. The test code remains completely unchanged because we have hidden the detail about submitting the form behind an abstraction.
* Refactoring is quite a hard thing to do and comes with experience. Sometimes it’s not worth removing 2 lines of duplication because it reduces the readability and clarity of the code. Refactoring is another subject in it’s own right.
* It certainly is the most neglected discipline in development. People tend to blame time pressures saying they don’t have time to refactor the code because they might break it. This is one of the areas where TDD really comes into it’s own. If we have a comprehensive test suite that we have faith in then we can refactor in the knowledge that our tests will catch any mistakes we might make.

## What Makes a Good Test?

* Asserts only one thing
  + Only one reason to fail
  + Easier to reason about the test
  + Quicker to get to pass
  + Keeps cost of test down
* Has a good name
  + Describes the behavior under test
  + Not necessarily the method or function under test
* Repeatable
  + Can be run again without manual intervention to clear anything down
* Fails helpfully
  + If the test fails it should indicate clearly why it failed, i.e. expected status to be X for invalid message.

## Different Flavours of TDD

* Behavior Driven Development (BDD)
  + Based on Dan North’s experience with TDD
  + Given/When/Then
  + Generally I haven’t seen BDD work that well wherever it has been applied.
  + Tend to be slow, brittle and get ignored when they break.
  + They are also expensive to get up and running in terms of the amount of effort to get the code behind the scripts running and then the effort involved in keeping the language at the right level, i.e. not too detailed but not too vague.
* Acceptance Test Driven Development (ATDD)
  + Take acceptance tests one at a time
  + Normal TDD fleshes out the implementation classes
  + Once test passes functionality is complete

## Further Reading

* Test-Driven Development by Example – Kent Beck
* Growing Object-Oriented Software Guided by Tests (GOOS) – Steve Freeman & Nat Pryce
* Is TDD Dead? Conversations - <http://martinfowler.com/articles/is-tdd-dead/>