Behaviour Driven Development

Test-Driven Design

BDD

TDD

Test-Driven Development

Test-First Programming

Three Rules of TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

" Now most programmers, when they first hear about this technique, think: *"This is stupid!" "It's going to slow me down, it's a waste of time and effort, It will keep me from thinking, it will keep me from designing, it will just break my flow."* However, think about what would happen if you walked in a room full of people working this way. Pick any random person at any random time. A minute ago, all their code worked.

Let me repeat that: *A minute ago all their code worked!* And it doesn't matter who you pick, and it doesn't matter when you pick. **A minute ago all their code worked!**

http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

Benefits

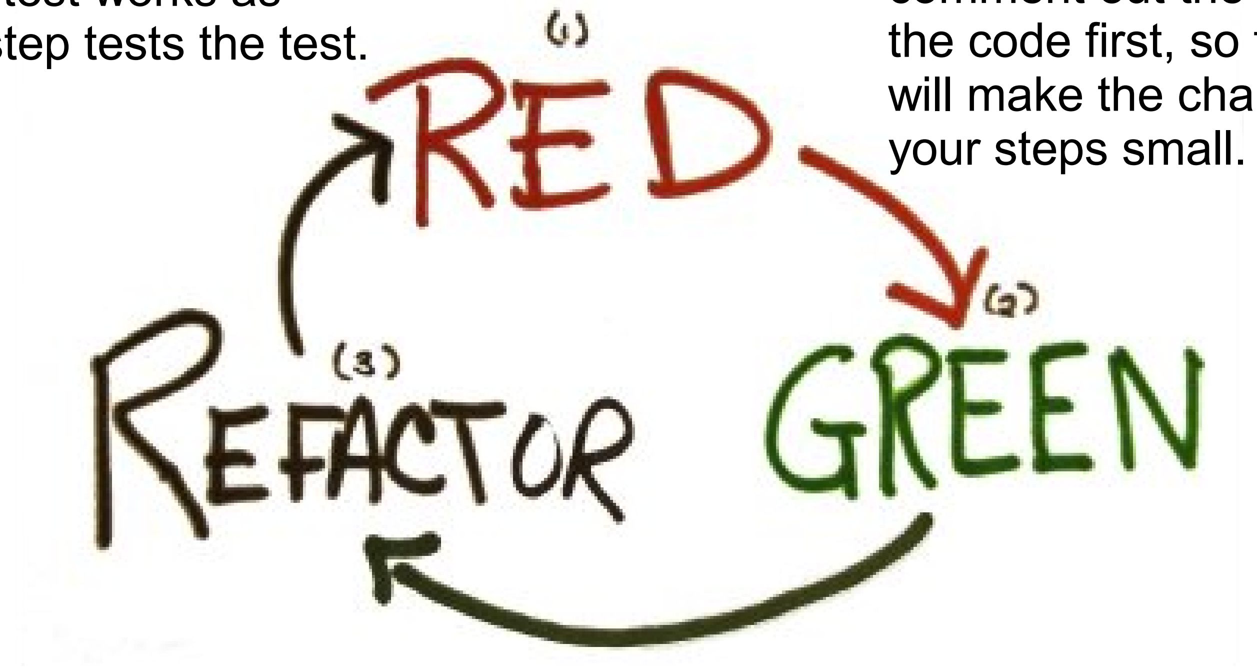
- **Better focus**: First think about **what** you want and write it down as a test. Afterwards think about **how** to implement it.
- **Immediate feedback** of your changes – change a line, run all tests, after a couple of seconds you will know whether anything was broken or whether everything works.
- Debuggers are needed very rarely – it's faster to undo your changes to the latest working state (~5 min ago).
- Once fixed bugs will not reappear.
- Tests are documentation – code samples of API usage.
- **Fearless change!** You can change code that your predecessor wrote. If you break something, you will know it in seconds.
- **Better design. Fun!** For code to be testable, it needs to be decoupled, so TDD makes you do more design. Also, writing tests first makes you think more about how the API is used.

http://blog.objectmentor.com/articles/2008/07/21/tdd-is-how-i-do-it-not-what-i-do
http://blog.objectmentor.com/articles/2008/06/24/so-you-want-your-code-to-be-maintainable
http://blog.objectmentor.com/articles/2008/02/16/the-quality-of-tdd
http://anarchycreek.com/2009/05/26/how-tdd-and-pairing-increase-production/
http://onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html

(1) Write one failing test.
- Run all tests and see it to fail, to be sure that the test works as expected. This step tests the test.

(2) Do the simplest change to make the one failing test pass.
- If the change is not simple, comment out the test and refactor the code first, so that the new design will make the change simple. Keep your steps small.

Cycle time:
max 5-10 min



(3) Refactor the code – improve its structure without changing its external behaviour. *This is the hardest step. Good design skills are needed!*
- Run tests after every change to make sure that the refactoring did not affect the code's behaviour. If it's been over 5 minutes since all tests passed, you have entered "refactoring hell" and you should revert to the last known working state (without version control your last known working state is an empty disk).

http://agileinaflash.blogspot.com/2009/02/red-green-refactor.html

Demo: Bowling Game Kata

Bowling Game Kata

Object Mentor, Inc.
www.objectmentor.com
blog.objectmentor.com

fitnesse.org
XProgramming.com
JU•org
www.junit.or

Copyright © 2005 by Object Mentor, Inc
All copies must retain this page unchanged.

http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
(Demo: ~20 min)

Testing or Design?

- For developers new to TDD, it may be easier to think of TDD first as a testing technique. Use TDD to implement a pre-existing design. ("Test-First Programming")
  - Here TDD lets you bring your defect rates down.
- When defects are in control, keep your eyes open and **listen to the code**. If some code or test is hard to write, there is an opportunity for you to learn something about the way the code is written. Maybe the design is wrong and you need to improve it. ("Test-Driven Design")
  - Now TDD has become a design technique.
  - Try using TDD for 6 months *without any up-front design*, to see with how little you can get away with.

http://programmingtour.blogspot.com/2009/07/test-first-and-test-driven-conversation.html (39 min)

TDD's Learning and Adoption

" Experienced practitioners, particularly those that have been involved in helping other developers learn the practice, note a life-cycle to TDD's learning and adoption:

1) The developer starts writing unit tests around their code using a test framework like JUnit or NUnit.
2) As the body of tests increases the developer begins to enjoy a strongly increased sense of confidence in their work.
3) At some point the developer has the insight (or is shown) that writing the tests before writing the code, helps them to focus on writing only the code that they need.
4) The developer also notices that when they return to some code that they haven't seen for a while, the tests serve to document how the code works.
5) A point of revelation occurs when the developer realises that writing tests in this way helps them to "discover" the API to their code. TDD has now become a design process.
6) Expertise in TDD begins to dawn at the point where the developer realizes that TDD is about defining behaviour rather than testing.
7) Behaviour is about the interactions between components of the system and so the use of mocking is fundamental to advanced TDD.

We have observed this progression in many developers, but unfortunately while most, with a little help, find their way to step 4, many miss the big wins found at steps 5, 6 and 7.

http://behaviour-driven.org/Introduction

What Test Should I Write?

- TDD is all about **specifying behaviour**.
  - Behaviour Driven Development: "BDD is TDD done right."
- To know what behaviour to specify next (by writing it in the form of a test), ask the question: **"What's the next most important thing the system *doesn't* do?"**
- *Test method names should be sentences* that read like a specification of the behaviour that is being specified by that particular test.
  - See this course's exercises for lots of examples.
- *Start with a trivial test.* It gets your started (and might cover an important corner case).
  - If summing all values in a list, first sum an empty list.
  - If writing a Tetris game, first test that the board is empty.

http://dannorth.net/introducing-bdd
http://techblog.daveastels.com/files/BDD_Intro.pdf

Expected-Actual Format

(works maybe 9 times out of 10)

```
expected = 6
actual   = sum(1, 2, 3)
assert expected == actual
```

Given-When-Then Format

(works maybe 9 times out of 10)

Given some initial context (the givens),
When an event occurs,
Then ensure some outcomes.

Scenario 1: Account is in credit
Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned

Scenario 2: Account is overdrawn past the overdraft limit
Given the account is overdrawn
And the card is valid
When the customer requests cash
Then ensure a rejection message is displayed
And ensure cash is not dispensed
And ensure the card is returned

http://dannorth.net/introducing-bdd

Principles

- Do The Simplest Thing That Could Possibly Work
- YAGNI – You Ain't Gonna Need It
  - "Always implement things when you **actually** need them, never when you just **foresee** that you need them."
  - "Even if you're totally, totally, totally sure that you'll need a feature *later on*, don't implement it now. Usually, it'll turn out either
    - a) you don't need it after all, or
    - b) what you actually need is quite different from what you foresaw needing earlier."
- When following TDD, changing the system afterwards is easy. Keep the system as simple as possible and add complexity/flexibility later when the need arises.

http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork
http://c2.com/cgi/wiki?YouArentGonnaNeedIt
http://en.wikipedia.org/wiki/You_Ain't_Gonna_Need_It

Course Material

- http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
- http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
- http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsKata
- http://agileinaflash.blogspot.com/2009/02/red-green-refactor.html
- http://dannorth.net/introducing-bdd
- http://techblog.daveastels.com/files/BDD_Intro.pdf
- http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork
- http://c2.com/cgi/wiki?YouArentGonnaNeedIt