Test Driven Development
TDD

Outline

ThoughtWorks®

- Good Unit Tests
- Discover TDD
- The TDD Rhythm
- Goals of TDD
- When to use TDD
- Pair programming
- Refactoring
- Q & A

Good Unit Tests…Kent Beck

ThoughtWorks®

- Run fast (they have short setups, run times, and break downs)
- Run in isolation (reordering possible)
- Use data that makes them easy to read and to understand
- Use real data (copies of production data) when they need to

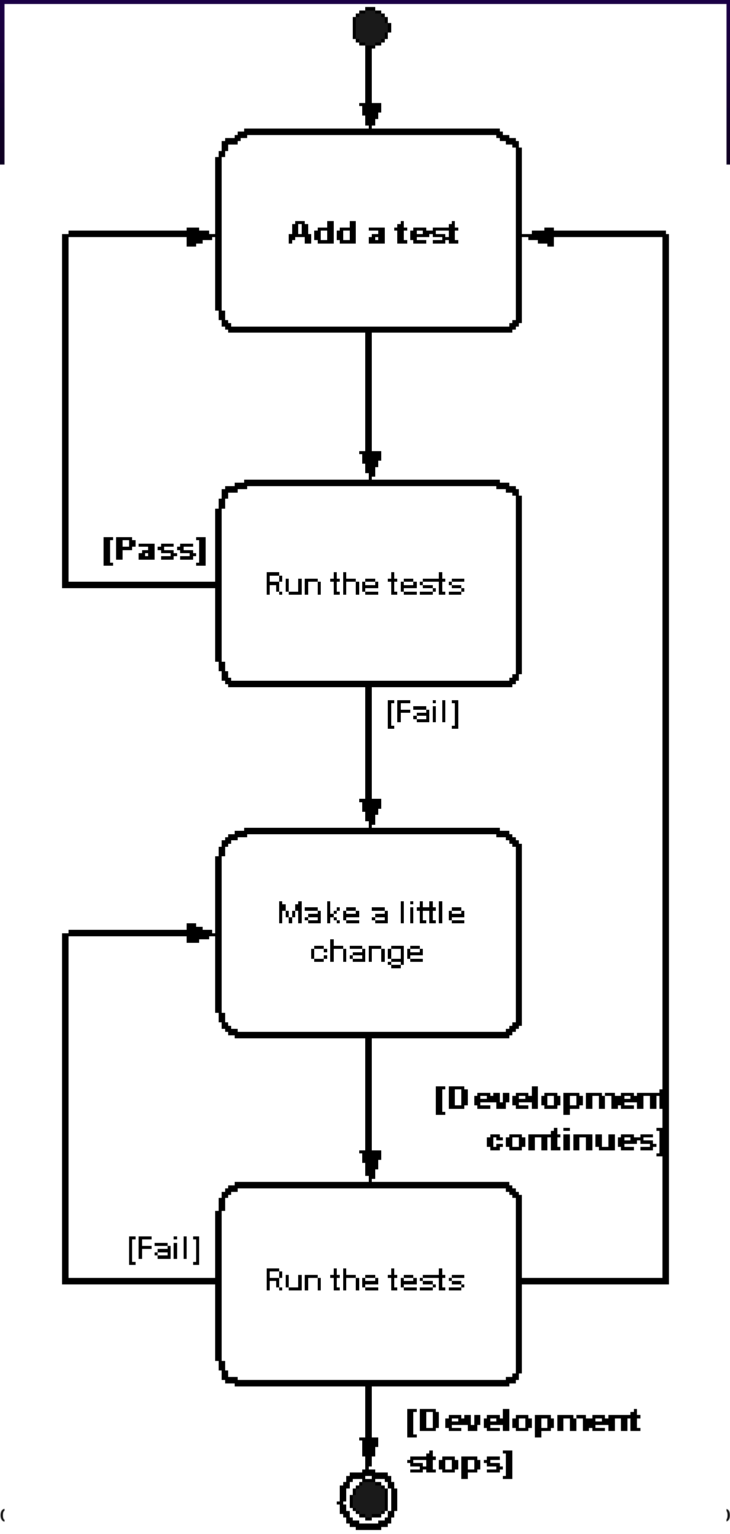
What is TDD?

ThoughtWorks®

- An iterative technique to develop software
- One must first write a test that fails before he writes a new functional code.
- The goals of TDD is specification and not validation
- A practice for efficiently evolving useful code

Overview

ThoughtWorks®



The TDD Rhythm is “Test, Implement, Refactor”

ThoughtWorks®

- Think about what a class ***should*** do
- Write a test for a method that will fail, but later will prove that the class fulfils its requirements
- Compile and run your test, getting the red bar
- Make the test pass, “faking” it where appropriate

The TDD Rhythm is "Test, Implement, Refactor"

ThoughtWorks®

- If possible write another failing test or assertion for the same method
- Make that test pass
- Repeat for all requirements of the method
- When all tests are green, refactor to remove duplication and simplify the design of the code

TDD is about Design, not Testing!

ThoughtWorks®

- Use TDD to produce the simplest thing that works (but not the dumbest!)
- Drive the design of the software through **unit tests**
- Focus on writing simple solutions for today’s requirements
- Write just enough code to make the tests pass, and no more
- Executable code becomes your requirement

Clean code that works

ThoughtWorks®

How does TDD achieve this?

- Predictable – Tells you when you are done
- Learn – Teaches you all lessons that the code has to teach
- Confidence – Green bar gives you more confidence
- Documentation – Good starting point to understand code

Clean code that works…

ThoughtWorks®

- Protection – Puts a test-harness around your code
- Avoids integration night-mares
- Automated test suit for you application

"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away." – [C&B – Eric]

When should I use TDD?

ThoughtWorks®

- Always!
- Write tests for anything you feel that might break
- Design of production code should always be test-*driven*
- No need to write tests for APIs you don't own

Two fundamental TDD rules (Kent Beck)

ThoughtWorks®

- Never write a single line of code unless you have a failing automated test.

- Eliminate duplication

Legacy Code

ThoughtWorks®

What do you do if you have a body of existing code without tests?

- Run away
- Write tests in the areas where you are changing the system
- If you are working on a defect, write a test to show the defect, then fix it.

When do I stop?

ThoughtWorks®

- The system works – All the tests pass
- Code communicates what it’s doing
- There is no duplicate code
- The system should have the fewest possible classes and methods

Smells that indicate TDD has gone wrong

- Testing private/protected methods
- Responsibility-laden objects
- Extensive setup/teardown
- Brittle tests
- Slow tests

Pair Programming

Advantages of Pair Programming

- Promotes better **communication** among the team members
- Brings out better **quality** of code
  - code-review
  - early defect detection and defect prevention
  - *Mentorship and "Pair-Learning"*
- Facilitates a smooth and gradual **induction** of new members to a team
- Improves retention and **confidence**
- Helps in spreading the **knowledge** about every part of a system to more than one person
- People **enjoy** themselves more

Refactoring improves design

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" - MartinFowler

Refactoring examples

ThoughtWorks®

Smell	Description	Refactorings
Comments	Should only be used to clarify "why" not "what". Can quickly become verbose and reduce code clarity.	Extract Method Rename Method Introduce Assertion
Long Method	The longer the method the harder it is to see what it’s doing.	Extract Method Replace Temp with Query Introduce Parameter Object Preserve Whole Object Replace Method with Method Object
Long Parameter List	Don't pass in everything the method needs; pass in enough so that the method can get to everything it needs.	Replace Parameter with Method Preserve Whole Object Introduce Parameter Object
Divergent Change	Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class	Extract Class

Retrospective

ThoughtWorks®

Pointers

ThoughtWorks®

- Kent Beck, Test Driven Development By Example.
- Test Infected - http://junit.sourceforge.net/doc/testinfected/testing.htm
- http://www.artima.com/intv/testdriven.html
- http://www.opensourcetesting.org/
- http://c2.com/cgi/wiki?WhatIsRefactoring
- http://www.refactoring.com/
- http://pairprogramming.com/

Questions

ThoughtWorks®

Thank You!

Naresh Jain
nashjain@gmail.com
http://jroller.com/page/njain

Anand Joglekar
ajoglekar@thoughtworks.com