SHARE
Technology · Connections · Results

JUnit and Test Driven Development: Why and How in Enterprise Software SHARE Orlando, February 2008

Slides by Justin Gordon
Architect
IBM, WebSphere Product Center
San Francisco, CA

Premise

“A comprehensive suite of JUnit tests is the single most import artifact created in a software project because it reduces bugs, facilitates adding new developers, and enables refactoring and performance tuning with confidence. Test-driven development (TDD) is the best way to build a suite of tests.”

Justin Gordon, Brisbane, CA
Trigo (WPC) Worldwide Headquarters, 2004

Roadmap

- **A tale of two development groups**
- JUnit Tests
- Test Driven Development
- Tools
- TDD Architecture
- Exercise TDD Mock Objects
- Getting Started

A Tale of 2 Development Groups

CONVENTIONAL

Architects → High Level Design (HLD) → Detailed Technical Design (DTD) → Coding → QA & Bug Fixing → Regressions → More QA & Bug Fixing → Major Release → Bug Fixing → Regressions → Bug Fixing → Minor Release → …

➔Painful mess! Unhappy developers, unhappy customers, unhappy managers!

AGILE

Stories and Requirements → Simple Specs → Write JUnit (automated) tests in Conjunction with Source → Ensure code coverage → Refactoring to make code *better* → QA → Limited bug fixing with JUnit tests for each bug fixed → Almost no regressions! → Performance tuning with confidence → Release → Very few bugs

➔ Happy developers, happy customers, happy managers!

Roadmap

- A tale of two development groups
- **JUnit Tests**
- Test Driven Development
- Tools
- TDD Architecture
- Exercise TDD Mock Objects
- Getting Started

JUnit Tests: What?

- **JUnit test**: a method, written in Java, that verifies the behavior of an individual unit of code, or occasionally of a larger subsystem, and **reports errors in an automated fashion**.

```
public void testAddReturnsSum() {
        int sum = Calculator.add(2, 3);
        assertEquals(5, sum);
}
```

JUnit Tests: Why?

- If JUnit tests pass and code coverage is high → **Nearly Bug-Free Code**!
- When JUnit tests cover requirements and tests pass → Code is Complete!
- JUnit tests facilitate automatic test running to **detect regressions instantly** during bug fix cycles. Can't do that with manual QA! Can't do that with QA Automation tools!

- Enables Courage and Creativity
- Developers (experienced and new) can change the code with confidence, enabling
    - **Refactoring**
    - **Performance Tuning**
    - JUnit tests serve to document and demonstrate the API
- Large team sizes, offshoring, complexity

Catch 22: Why not write JUnit tests?

- "Normal" development cycle inhibits JUnit test creation
- Catch-22: existing quality is low, so developers are too busy fixing problems found in the field to write tests.

- (Bad) Attitude: "It's QA's job to find my bugs. I don't have time to write tests."
- Skills: tough to learn how to write JUnit tests for new code. Many new patterns!
- Even tougher for old code!
- Unless existing code is designed for testability, implementing JUnit tests is very **difficult**.

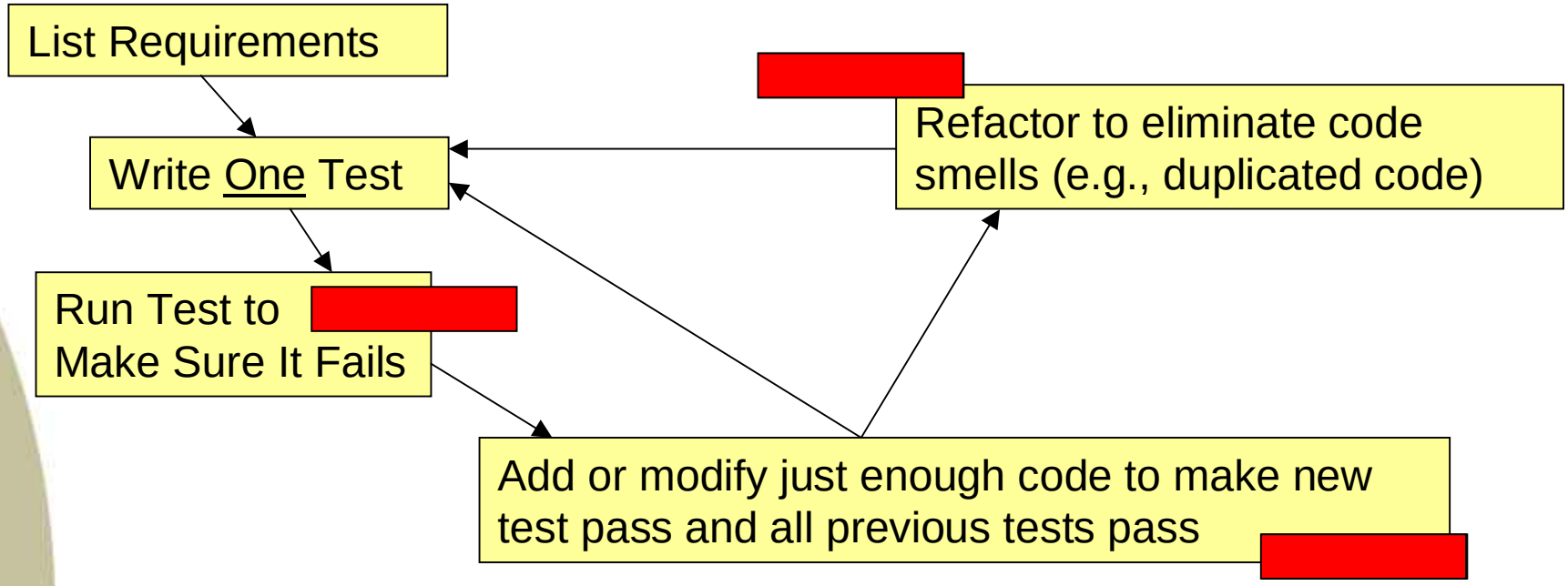
Really Tough!

Roadmap

- A tale of two development groups
- JUnit Tests
- **Test Driven Development**
- Tools
- TDD Architecture
- Exercise TDD Mock Objects
- Getting Started

What is Test Driven Development (TDD)?

"Programming practice in which **all** production code is written in response to a failing test."



Read Kent Beck's:
"Test Driven Development By Example"

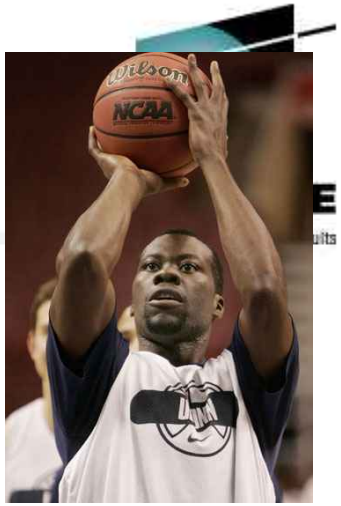
What Is Not Test Driven Development?

- Any time you write code that is not fixing a failing test.
- I.e., Writing code, then writing tests or intending to eventually write tests.
- Relying on QA to automate their manual tests.
- Be honest when trying this.
- Conventional Big Up Front Design is not TDD!

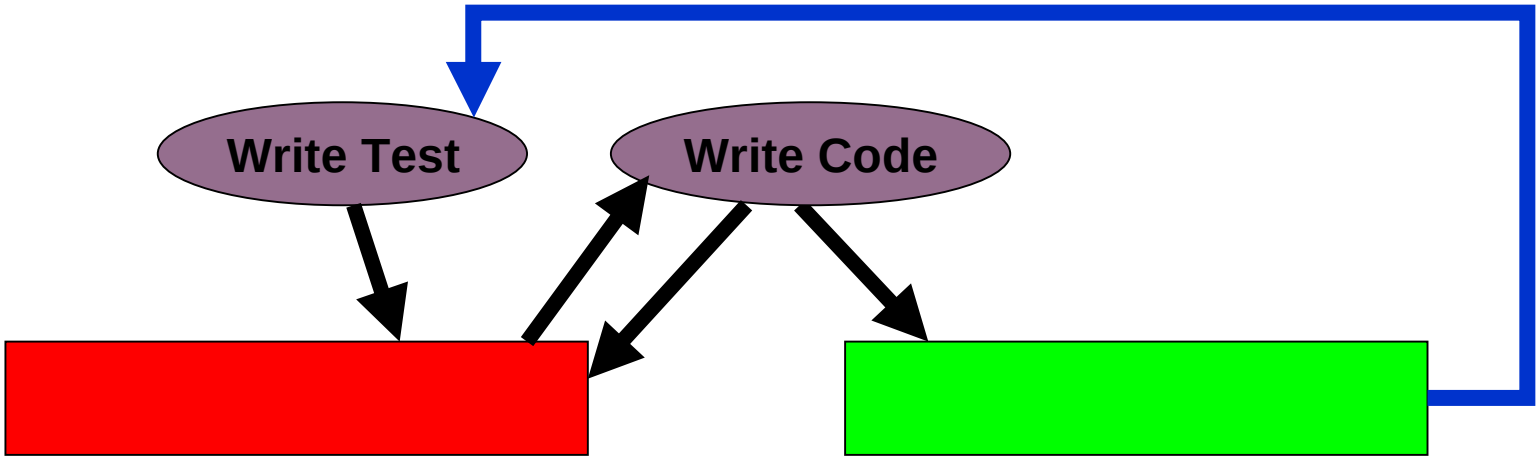
Why TDD → Code Coverage & Better Code

- Guarantees existence of JUnit tests covering most, if not all of your code!
- Guarantees code will be written to be testable.
  - Reverse is also true: if you write your code first, and then your tests, you may have difficulty writing tests for the new code, and then you may not write the tests at all! More natural to write untestable code unless tests written at the same time.
- Solves the **motivation problem**. Test writing becomes part of the coding process, not a tedious afterthought.
- Produces better code: more **decoupled**, with **clearer**, **tighter contracts**.

Shooting hoops? Practice Makes Perfect



- **Developers improve skills** because of the immediate feedback from the tests, rather than months later from QA!



- Academic study confirms **higher quality code**: "An Initial Investigation of Test Driven Development in Industry" by Boby George and Laurie Williams, 2003, http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf
- TDD developers took more time (16%), but non-TDD developers did not write adequate automated test cases even though instructed to do so!

Roadmap

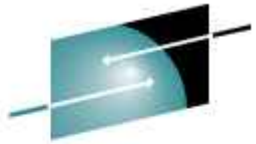
- A tale of two development groups
- JUnit Tests
- Test Driven Development
- **Tools**
- TDD Architecture
- Exercise TDD Mock Objects
- Getting Started

Making it Happen: Tools for Success

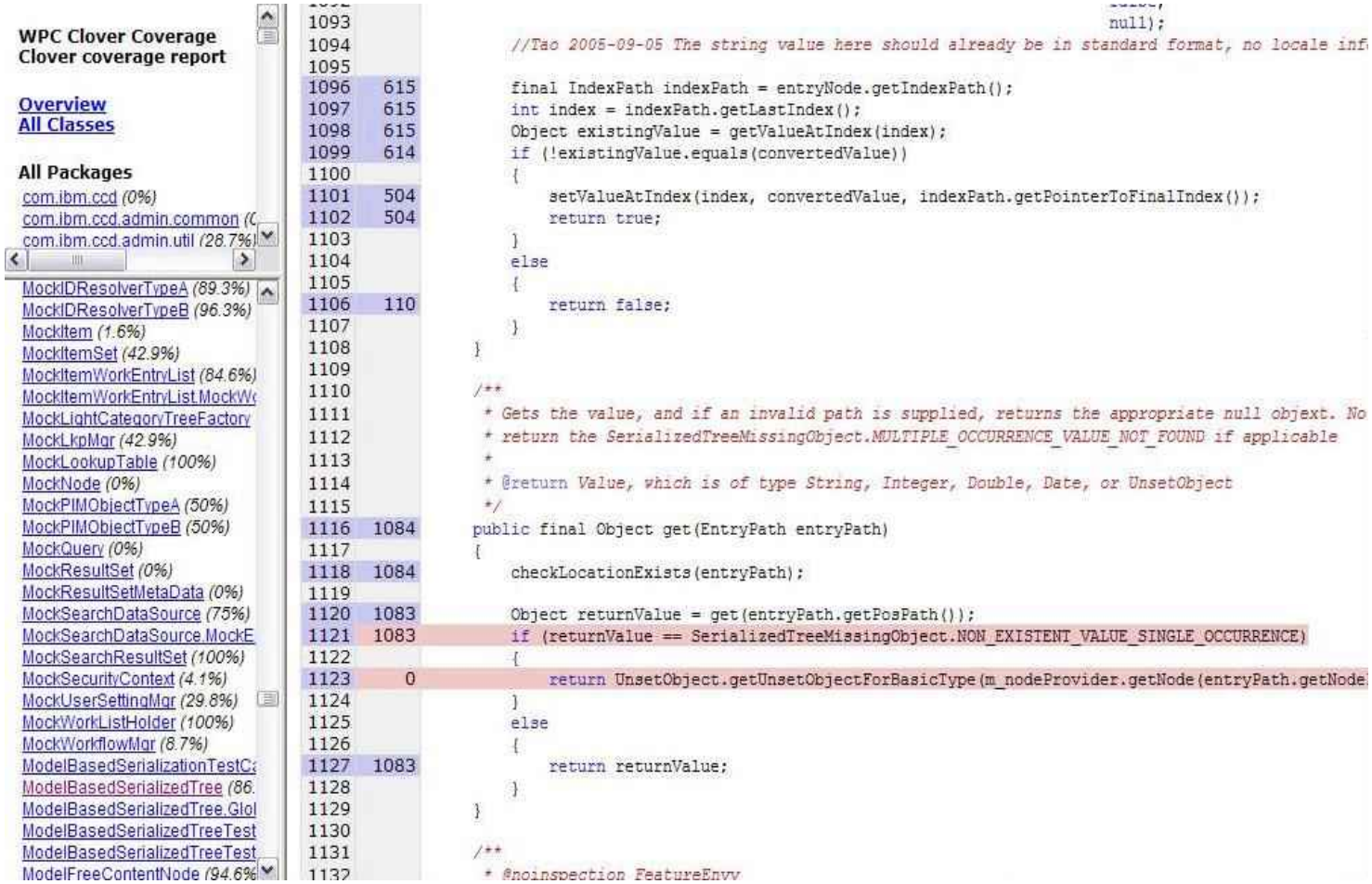
SHARE
Technology · Connections · Results

- IDEs: Eclipse and IntelliJ offer these essentials:
  - Ease of use to run a single new test
  - Refactoring tools.
- Code Coverage
  - **Clover** or **Emma** are the most popular.
  - How do you know how good your tests are?
  - Measure progress for morale (and management reports).
- Cruise Control
  - Automated system for building code, running JUnit tests and reporting on code coverage.
  - Alerts team of issues (build or JUnit) within minutes of checkins

Sample Clover Coverage HTML Report



The pink line in the lower right indicates that line 1121 was executed 1083 times and it always evaluated to FALSE, demonstrated by the zero in line 1123. The challenge is to write a unit test that hits line 1123. That is the game!



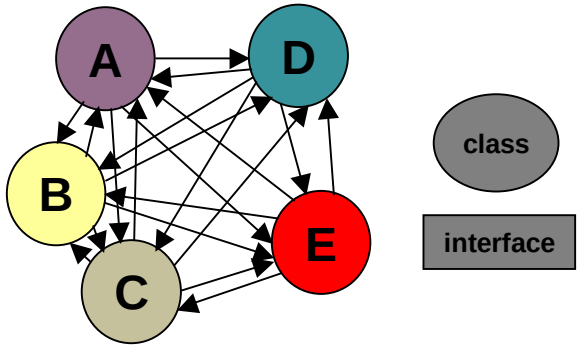
Roadmap

- A tale of two development groups
- JUnit Tests
- Test Driven Development
- Tools
- **TDD Architecture**
- Exercise TDD Mock Objects
- Getting Started

Architectural benefits of TDD

- **Without TDD**, you often see **tight coupling** between classes, making the implementation of new tests prohibitively **painful.**

Tight Coupling!

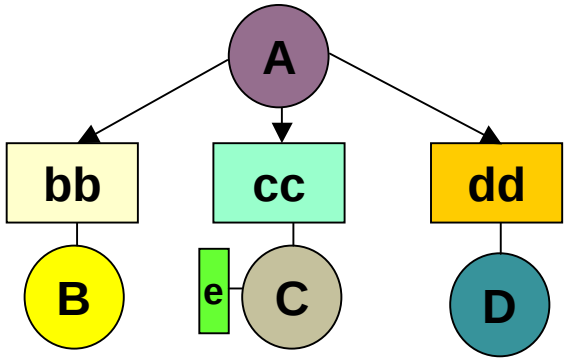


- **With TDD**

- Better abstractions and better decoupling of classes.

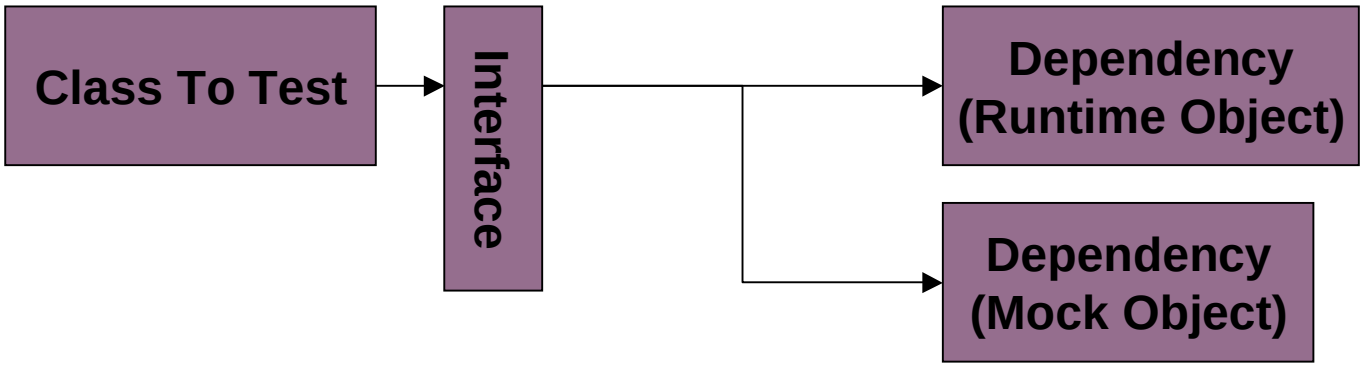
- **Loose coupling**, otherwise impossible to test individual components.

Loose Coupling: TDD!



Making it Happen: Isolating Code for Testing

- Successful TDD depends on **dependency isolation** – you need to separate the code to be tested from the rest of the system.
  - This is THE main technical challenge, esp. for database and integration points
  - Must decouple classes with interface/implementation/mock object pattern
  - Use a combination of **dependency location** and/or **injection** for dependencies (following slides)
  - Typically mocking out some more complicated resource, such as an object that typically references the DB with a replacement **mock object**
  - You almost cannot do TDD without using mock objects.
  - Tip: minimize business logic in mock objects because you have to duplicate that logic in the real implementation. Refactor code to minimize business logic in mock classes.



Roadmap

- A tale of two development groups
- JUnit Tests
- Test Driven Development
- Tools
- TDD Architecture
- **Exercise TDD Mock Objects**
- Getting Started

Exercise: Fibonacci Numbers

Objective:

Write an application to calculate Fibonacci numbers using JUnit and Test Driven Development methodology

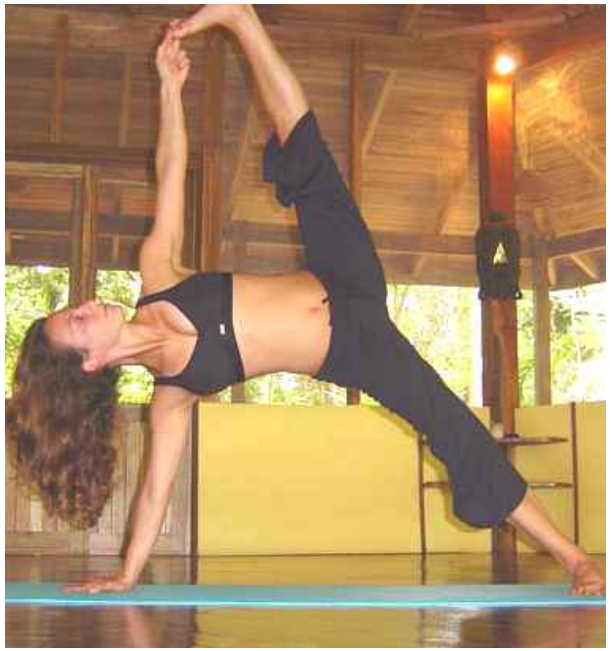
Roadmap

- A tale of two development groups
- JUnit Tests
- Test Driven Development
- Tools
- TDD Architecture
- Exercise TDD Mock Objects
- Database Techniques: DOF
- **Getting Started**

Making it Happen: Dealing with Legacy Code

- If only we could write everything from scratch again!
- Expect islands of old, untested, and untestable code
  - UI code tends to be particularly problematic
  - Most legacy code will be essentially untestable
- Try piecewise remodeling
  - Discard old modules one-by-one, replacing with TDD code
- Try "en**crap**sulation"
  - Wall off legacy junk behind a façade interface (if possible; sometimes not)
  - Mock out legacy code when testing new modules

Making it Happen: Getting the Team Started



- **Get Beck's "Test Driven Development", start up your IDE and do TDD! Or try to recreate my accounting example.**
- **Solidify commitment at every level of the organization**
  - TDD slower for first 6 months; net speedup afterwards.
  - Don't expect to hold the same schedule and "just add testing"!
  - Systematically discover and eliminate obstacles.
  - TDD takes discipline. Align incentives, communication, work environment – *everything*
- **Start with a new project**
  - TDD can be done against existing code, but MUCH harder
  - Focus on lower levels of the system first
- **Allow extra time**
  - for **learning**: there are many new skills and patterns to pick up.
  - for **re-architecture**: existing architecture probably doesn't support testing
- **Expect discomfort at first** – developers not used to working this way.
  - Start with a few respected "early adopters" and a trial run

Conclusion

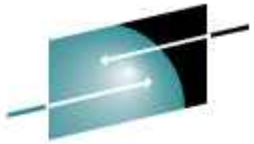
- How would you choose between a project with awesome JUnit tests and a project without JUnit, but lots of great architectural documents and other documentation? I'd take the JUnit one hands down.
- Documentation gets out of date quickly. Code without tests may be quite buggy, and even if it's not buggy, would I trust myself to join a project and not introduce bugs without JUnit?
- **Having a comprehensive suite of JUnit tests is the most import piece of intellectual property in a software project.**
- Why? First you have very few bugs. Second, developers, *new* and old, can change the code with confidence because they know immediately if they break something. This enables the two most important activities in a software project.
  - **Refactoring**
  - **Performance Tuning**
- And TDD is the best way to get that suite of tests!

Resources

- Just Do It! You cannot just read books on it! Just Do It!

- Use IntelliJ or Eclipse and try doing TDD on a simple example
- Books by Kent Beck – “Test Driven Development”, “Extreme Programming Explained”
- Books by Martin Fowler, especially “Refactoring: Improving the Design of Existing Code”

Appendices

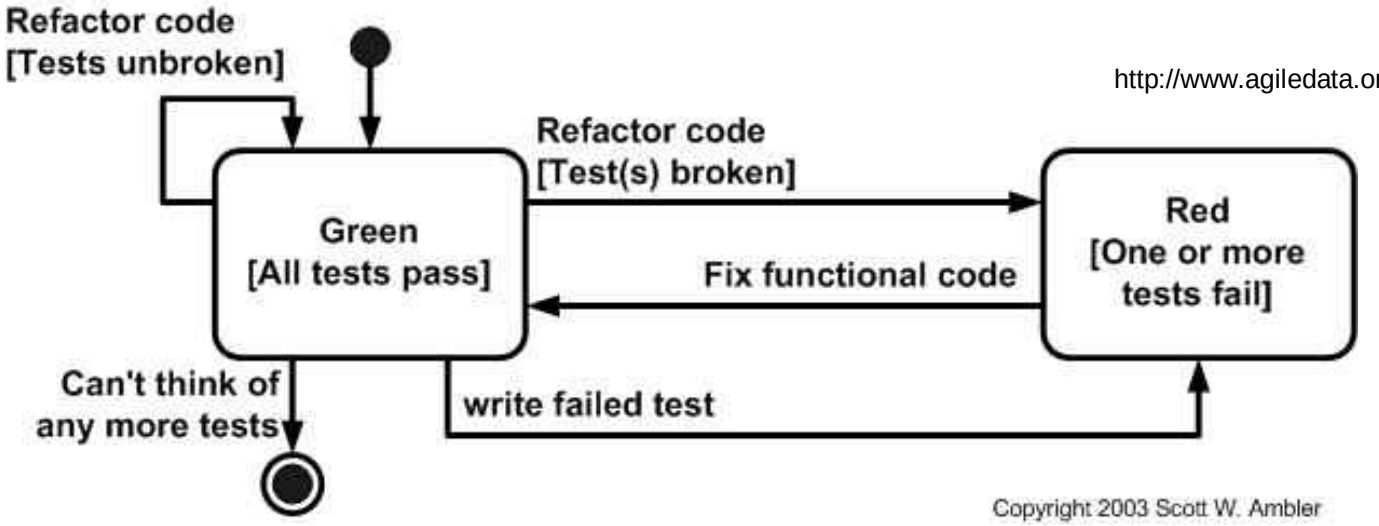


Patterns: Naming Test Methods

- test**{MethodName}**Returns**{Value}**When**{Condition}**
  - test**GetAmount**Returns**Zero**When**NoInvoicesAdded**
  - test**GetAmount**Returns**InvoiceAmount**When**SingleInvoiceAdded**
  - test**GetAmount**Returns**SumofMultipleInvoices**WhenMultipleInvoicesAdded
- test**{MethodName} {DoesSomething}**When**{Condition}**
  - test**AddInvoiceThrows**When**InvoiceLimitExceeded**
- Why such long method names?
  1. Method names print out in test failures
  2. Programmers never call these methods

What about Refactoring?

- Refactoring is the process of changing the code to make it of higher quality, for example by better naming, eliminating duplication, easier to read, etc.
- Idea is to get the unit tests working, even if the code is ugly, then keep improving code while keeping the unit tests passing.
- One implication is that it is more important to have a library of unit tests rather than the most beautiful “architectural” code and greatest documentation! → If the tests are good, we can manipulate (refactor) the code without fear.
- Read Martin Fowlers “Refactoring: Improving the Design of Existing Code”



http://www.agiledata.org/essays/tdd.html

Better Developers: A Star is Made

"And it would probably pay to rethink a great deal of medical training. Ericsson has noted that most doctors actually perform worse the longer they are out of medical school. Surgeons, however, are an exception. That's because they are constantly exposed to two key elements of deliberate practice: immediate feedback and specific goal-setting.

The same is not true for, say, a mammographer. When a doctor reads a mammogram, she doesn't know for certain if there is breast cancer or not. She will be able to know only weeks later, from a biopsy, or years later, when no cancer develops. Without meaningful feedback, a doctor's ability actually deteriorates over time."

STEPHEN J. DUBNER and STEVEN D. LEVITT, "A Star Is Made", New York Times, May 7, 2006:
http://www.nytimes.com/2006/05/07/magazine/07wwln_freak.html?pagewanted=2&ei=5070&en=7265a75e2cf70a87&ex=1147838400&emc=eta1

Implications: JUnit and *especially* TDD will groom better developers because they see the results of their code continually, rather than waiting for months.

Guidelines for Test First Design

- If time allows, I can cover these points, but this is really a one day seminar in doing Test Driven Development. Or maybe this would be fun to discuss at another brown-bag.
- Copied from http://xprogramming.com/xpmag/testFirstGuidelines.htm which is a summary of Beck's and Fowler's work.

1. The name of the test should describe the requirement of the code
2. There should be at least one test for each requirement of the code. Each possible path through of the code is a different requirement
3. Only write the simplest possible code to get the test to pass, if you know this code to be incomplete, write another test that demonstrates what else the code needs to do
4. A test should be similar to sample code, in that it should be clear to someone unfamiliar with the code as to how the code is intended to be used

Guidelines for Test First Design

5. If a test seems too large, see if you can break it down into smaller tests

6. If you seem to be writing a lot of code for one little test, see if there are other related tests you could write first, that would not require as much code

7. Test the goal of the code, not the implementation

8. One test/code/simplify cycle at a time. Do not write a bunch of tests, and try to get them working all at once

Guidelines for Test First Design

9. Keep writing tests that could show if your code is broken, until you run out of things that could possibly break

10. When choosing an implementation, be sure to choose the simplest implementation that could possibly work

11. If you are unsure about a piece of code, add a test you think might break it

12. A test is one specific case, for which there is a known answer

Guidelines for Test First Design

9. The name of the test should describe the requirement of the code

10. There should be at least one test for each requirement of the code. Each possible path through of the code is a different requirement

11. Only write the simplest possible code to get the test to pass, if you know this code to be incomplete, write another test that demonstrates what else the code needs to do

12. A test should be similar to sample code, in that it should be clear to someone unfamiliar with the code as to how the code is intended to be used

Guidelines for Test First Design

13. If all of the tests succeed, but the program doesn't work, add a test

14. Tests should be as small as possible, before testing a requirement that depends on multiple things working, write a test for each thing it depends

15. Tests should not take longer than a day to get working, typical test/code/simplify cycles take around 10 minutes

16. Don't test every single combination of inputs. Do test enough combinations of inputs to give you confidence that the any code that passes the test suite will work with every single combination of inputs

Guidelines for Test First Design

17. Do not write a single line of code that doesn't help a failing test succeed. (Clarification for GUI's, some aspects of GUI's are impossible to test automatically, so it will have to be an acceptance test that drives you two write some GUI code. Use automated testing whenever possible)

18. Do not fix a bug until you have written a test that demonstrates the bug

What is the simplest code?

1. All of the tests run
2. There is no duplicate code (any given code segment or structural pattern should appear "once and only once")
3. Clarity. The code and tests communicate the intent as clearly as possible
4. The code is minimal (no classes or methods unnecessary to get the tests to pass)

The Test-Code-Simplify cycle

1. Write a single test
2. Compile it. It shouldn't compile, because you haven't written the implementation code it calls
3. Implement just enough code to get the test to compile
4. Run the test and see it fail
5. Implement just enough code to get the test to pass
6. Run the test and see it pass
7. Refactor for clarity and "once and only once"
8. Repeat