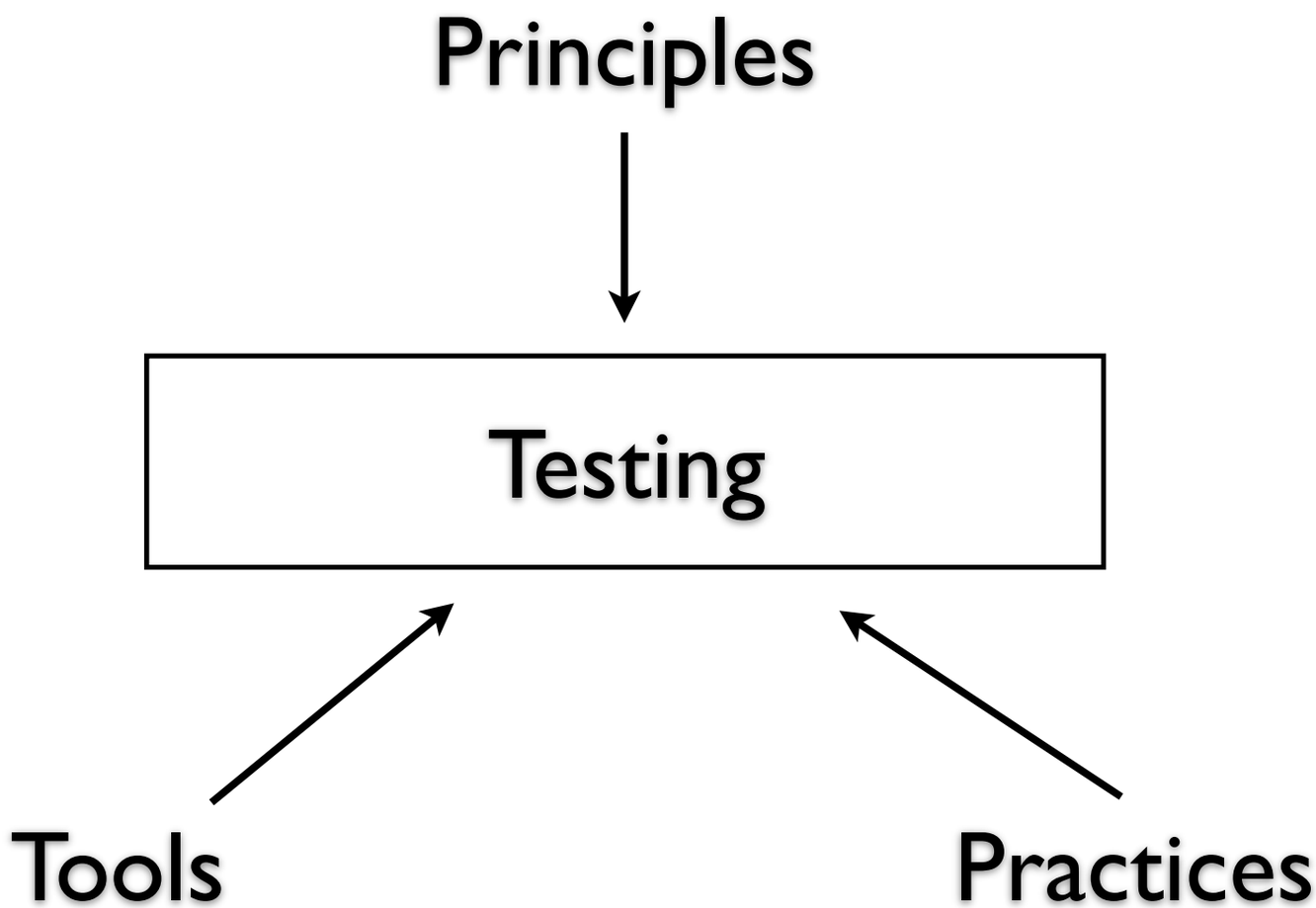
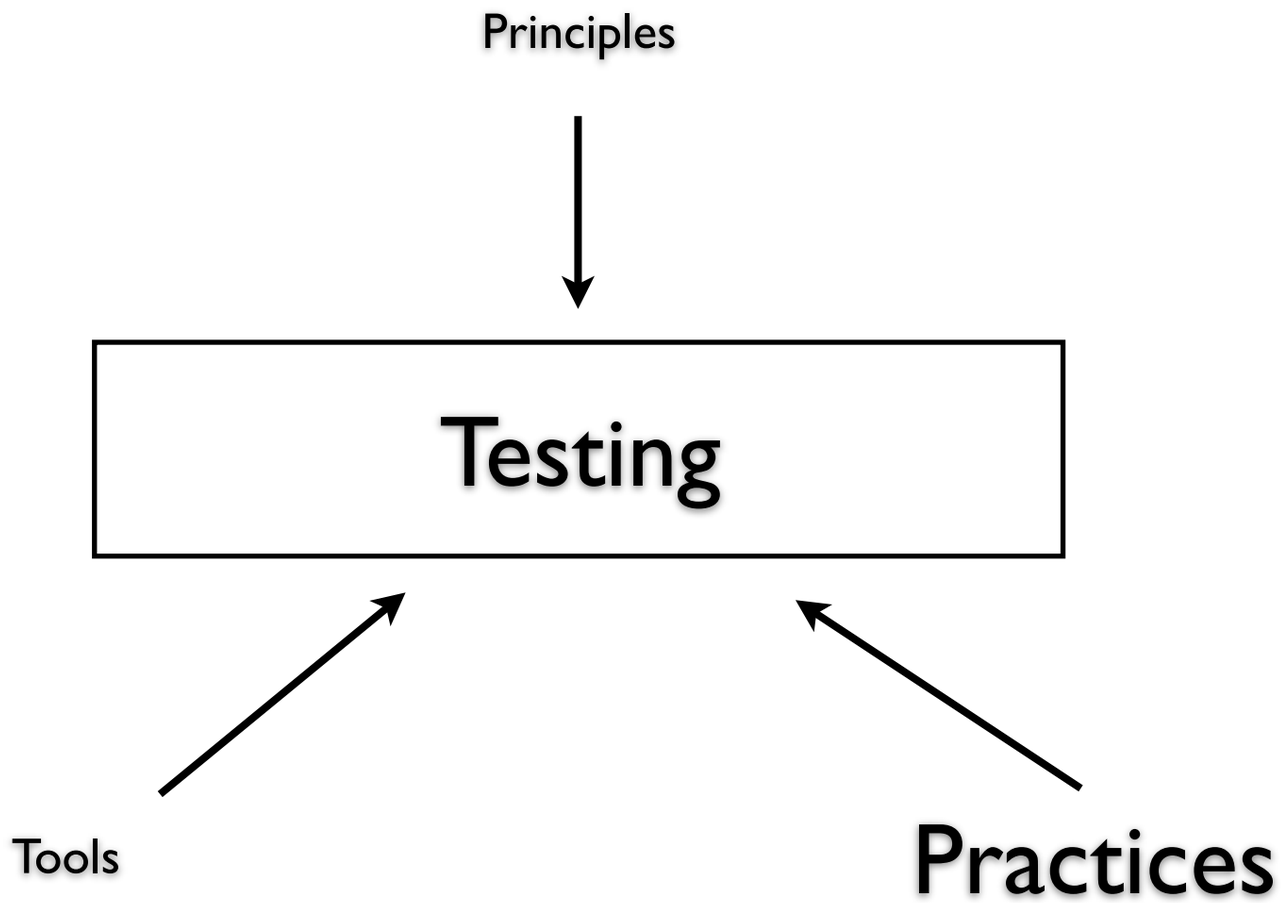
Test-Driven Development

Overview



Overview



Principles of Testing

Testing

- Testing can be performed in many ways:
  - Formal methods to reason about correctness, safety, etc.
  - Statistical analysis or simulation to investigate complex systems.
  - Random interaction with the system.

Manual Testing

- Perform some task with our system and check the output.
- Sometimes, we write programs this way.

Photo courtesy of University of Waterloo (http://www.cs.uwaterloo.ca/40th/images/image019.jpg)

Automated Testing

- Sometimes, test are automated, typically to reduce testing effort.
- Automated tests are programs...

Photo courtesy of 30gms (http://30gms.com/images/uploads/lazboy.jpg)

Testing Tools

Automated Testing

A system test, written in Cucumber:

```
Feature: Addition

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    When  I press add
    And   I enter 70 into the calculator
    Then  the screen should display 120
```

Automated Testing

A unit test written in Java with JUnit:

```
public cl

   @Test
   public void displayChangesWhenRegisterChanges() {
      Register register = new Register();
      Display display =

      register.cha

      String displa
      assertEquals("5", displayed);
   }
}
```

Annotation indicates that this method is a test

Check that the value produced by the program is what we expect

JUnit

General format for a JUnit test:

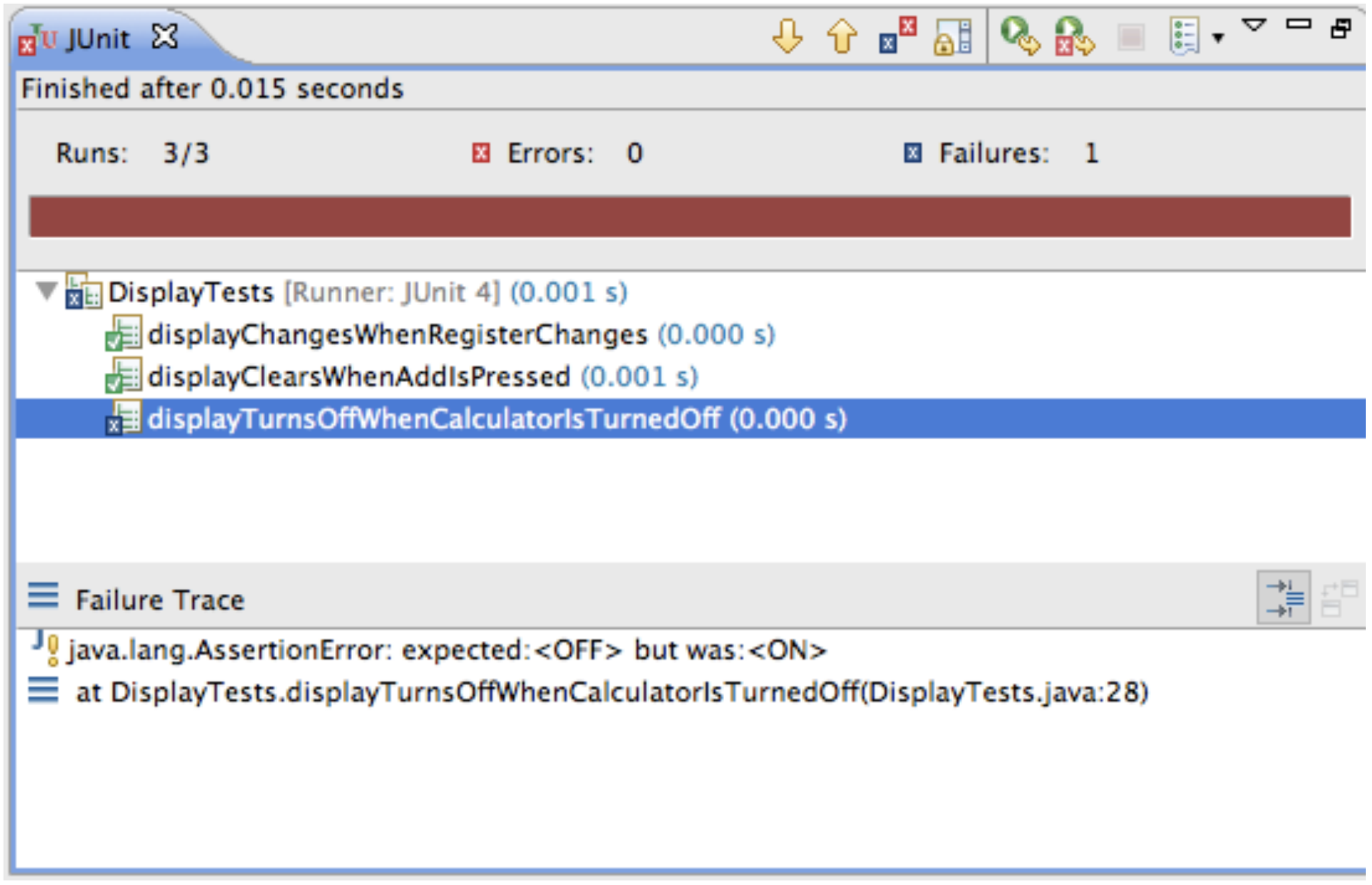
```
public class AnExampleTestCase {

  @Test
  public void aTestMethod() {
    // Do something to the system
    // Retrieve a value from the system
    // Check the value is as we expect
  }
}
```

JUnit

Related tests can be grouped in one class:

```
public class DisplayTestCase {

  @Test
  public void displayChangesWhenRegisterChanges() { ... }

  @Test
  public void displayClearsWhenAddIsPressed() { ... }

  @Test
  public void displayIsOffWhenCalculatorIsTurnedOff() { ... }

}
```

JUnit - Tools


JUnit
Finished after 0.015 seconds
Runs: 3/3
Errors: 0
Failures: 1
DisplayTests [Runner: JUnit 4] (0.001 s)
displayChangesWhenRegisterChanges (0.000 s)
displayClearsWhenAddIsPressed (0.001 s)
displayTurnsOffWhenCalculatorIsTurnedOff (0.000 s)
Failure Trace
java.lang.AssertionError: expected:<OFF> but was:<ON>
at DisplayTests.displayTurnsOffWhenCalculatorIsTurnedOff(DisplayTests.java:28)

Test-Driven Development (TDD)

Question...

- What difficulties do you have when programming?

Programming Pressures

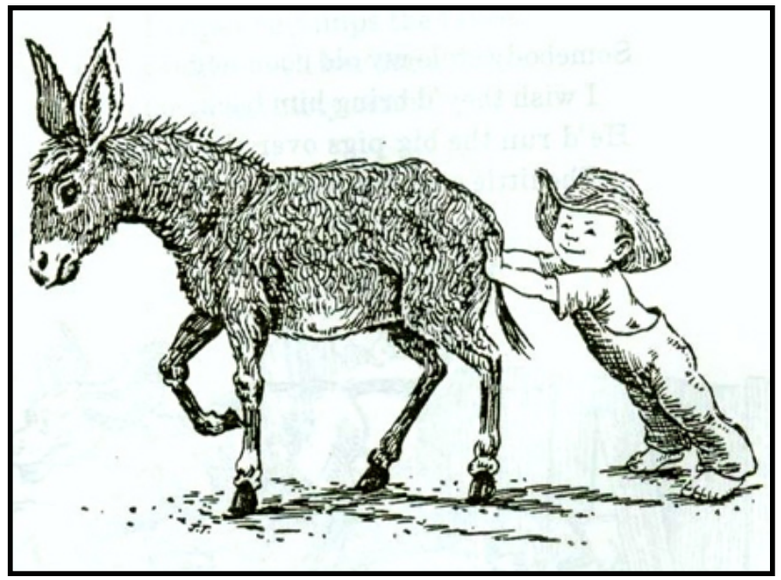
Anxiety:

How do I start to solve this problem?

Am I making progress?

Photo from http://futurity.org/wp-content/uploads/2009/12/anxious.jpg

Programming Pressures



Fear of change:

If I change this bit of the code, will I break something else?

Programming Pressures

Confusion:

What does this part of the code do again?

The code is a mess :(

Photo from http://www.altergroup.com/blog/wp-content/uploads/2010/02/maze.jpg

Programming Pressures

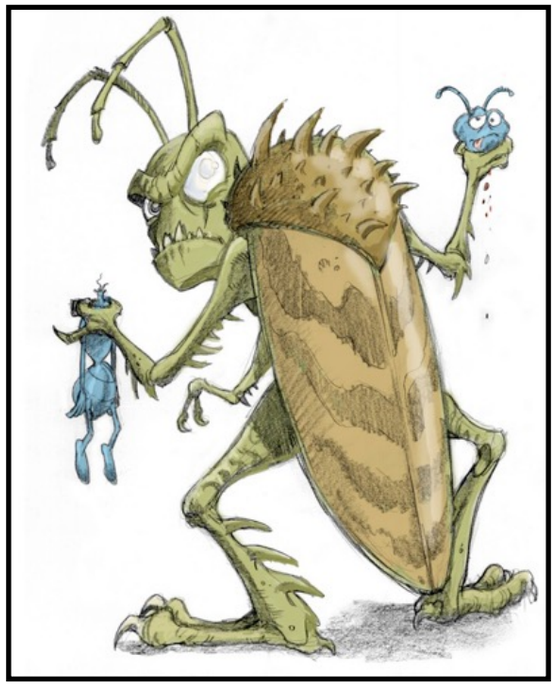
Despair:

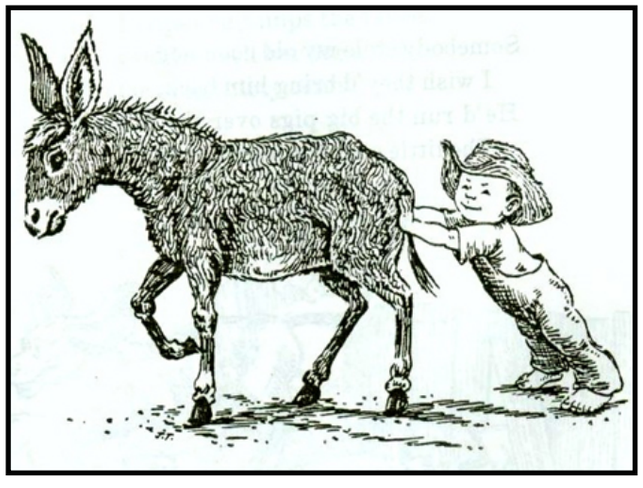
What's causing this bug?!

It's 4am :'(

Photo from http://www.digitaldreammachine.com/blogimages/luxo/BlueSkyPixarHopper.jpg

Test-Driven Development

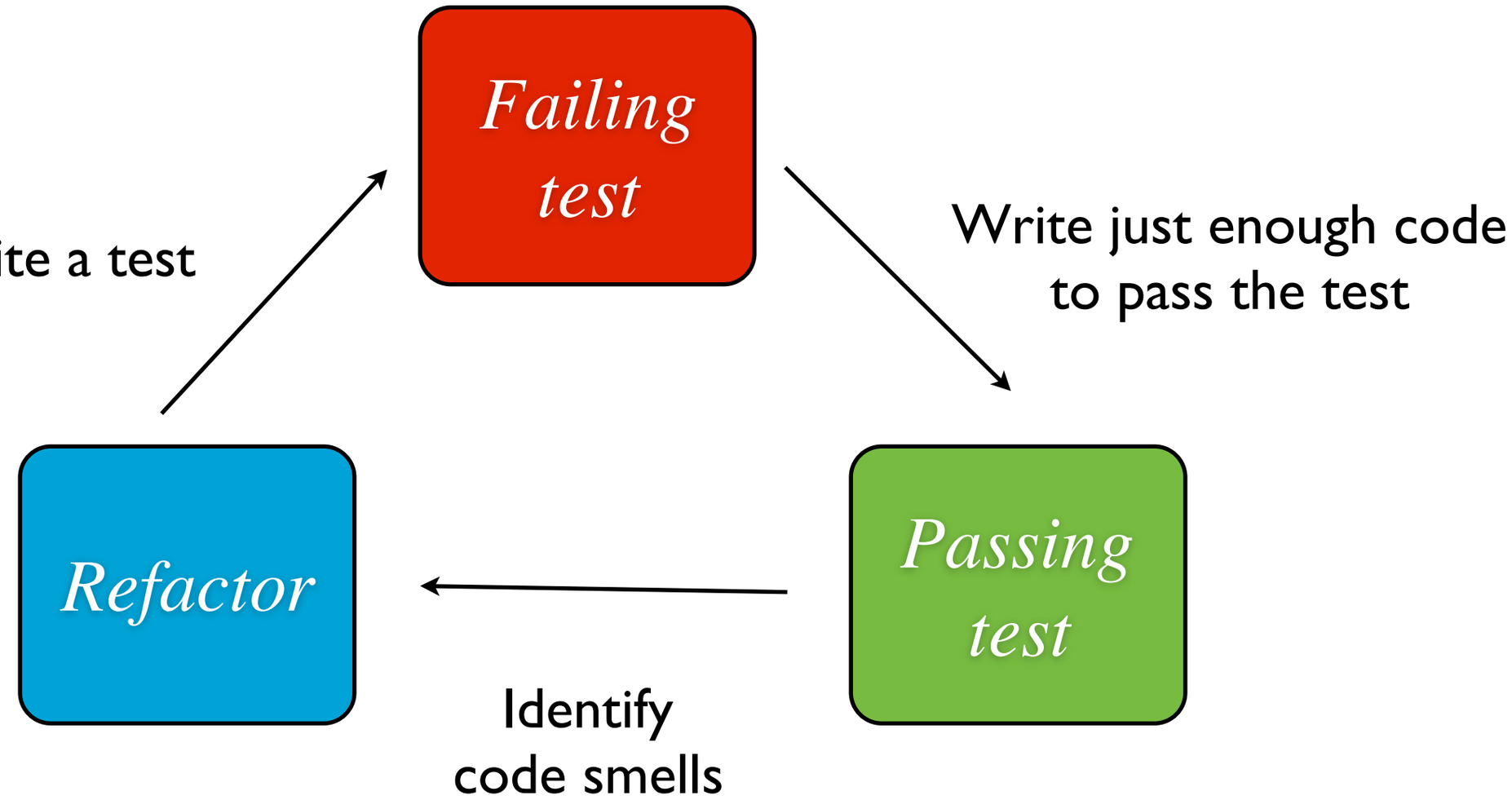




?

TDD is a very simple development process.
It might help with some of these issues.

Test-Driven Development



Refactoring

- Refactoring is a change to code that improves its structure, but does not affect its behaviour.
- We refactor to improve the quality of our code, to make it easier to understand, and to facilitate future changes.

Photo from http://embodyfitness.net/wp-content/uploads/2010/04/dirty-dishes1.jpg

Code smells

- Some examples:
  - **Duplicated code -** one change in requirements causes many changes to code
  - **Long methods -** harder to understand and harder to re-use
  - **Poor names -** fail to communicate intentions

    Many more: http://c2.com/xp/CodeSmell.html

TDD Example - Bowling

1	2	3	4	5	6	7	8	9	10
5 3	4 2	9 1	6						
8	14	30							

- Ten frames in each game.
- Two balls thrown in (most) frames.
- Bonus points for spares and strikes.

Example based on Agile Principles, Patterns and Practices In C#, Martin & Martin, 2006.

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BowlingTests {

    @Test
    public void scoreForOneFrameGame() {
        Game game = new Game();
        game.bowl(5, 3);
        assertEquals(8, game.score());
    }
}
```

Scoring Code

Failing

```
public class Game {

    public void bowl(int i, int j) {

    }

    public int score() {
        return 0;
    }
}
```

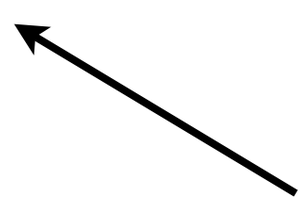
Fail: Expected **8**, got **0**.

Next step: Just enough code to pass the test.

Scoring Code

Passing

```
public class Game {

    public void bowl(int i, int j) {

    }

    public int score() {
        return 8;
    }
}
```



Pass: Expected **8**, got **8**.

Next step: Look for code smells.

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BowlingTests {

    public void scoreForOneFrameGame() { ... }

    @Test
    public void scoreForTwoFrameGame() {
        Game game = new Game();
        game.bowl(5, 3);
        game.bowl(4, 2);

        assertEquals(8 + 6, game.score());
    }
}
```

Failure: Expected **14**, but got **8**.

Next step: Just enough code to make it pass.

Scoring Code

Failing

```
public class Game {

    public void bowl(int i, int j) {

    }

    public int score() {
        return 8;
    }
}
```

Scoring Code

Passing

```
public class Game {

    private int total = 0

    public void bowl(int i,
        total += i + j;
    }

    public int score() {
        return total;
    }
}
```

i and j are not very descriptive names.

Pass: Expected **14**, got **14**.

Next step: Look for code smells.

Scoring Code

Passing

```
public class Game {

    private int total = 0;

    public void bowl(int firstThrow, int secondThrow) {
        total += firstThrow + secondThrow;
    }

    public int score() {
        return total;
    }
}
```

Refactor: Make parameter names descriptive.

Next step: Another test.

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Game {

    public void scoreForOneFrameGame() { ... }
    public void scoreForTwoFrameGame() { ... }

    @Test
    public void scoreForFirstFrameOfTwoFrames() {
        Game game = new Game();
        game.bowl(5, 3);
        game.bowl(4, 2);

        assertEquals(8, game.scoreForFrame(1));
    }
}
```

Scoring Code

Failing

```
public class Game {

    private int total = 0;

    public void bowl(int firstThrow, int secondThrow) {
        total += firstThrow + secondThrow;
    }

    public int score() { return total; }

    public int scoreForFrame(int frameNumber) {
        return 0;
    }
}
```

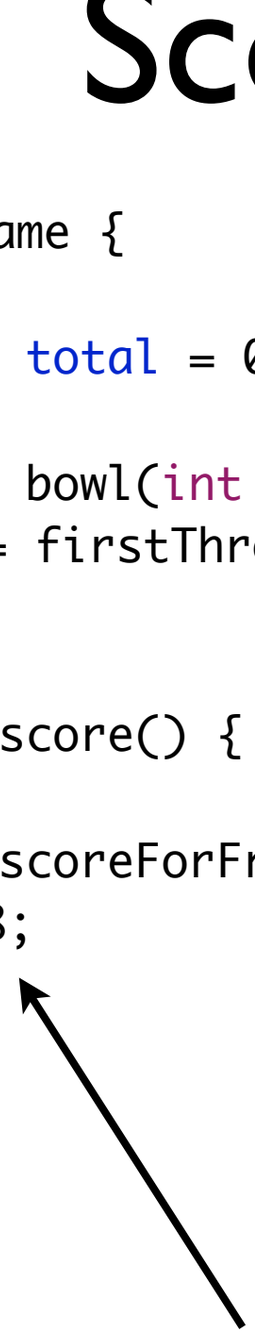
Failure: Expected **8**, but got **0**.

Next step: Just enough code to make it pass.

Scoring Code

Passing

```
public class Game {

    private int total = 0;

    public void bowl(int firstThrow, int secondThrow) {
        total += firstThrow + secondThrow;
    }

    public int score() { return total; }

    public int scoreForFrame(int frameNumber) {
        return 8;
    }
}
```



Pass: Expected **8**, got **8**.

Next step: Look for code smells.

Test Case

Passing

```
import static org.junit.Assert.*;
import org.junit.*;

public class BowlingTests {

    public void scoreFor
    public void score

    @Test
    public void score
        Game game = new Game();
        game.bowl(5, 3);
        game.bowl(4, 2);

        assertEquals(8, game.scoreForFrame(1));
    }
}
```

Every test method has to create a game object.

Test Case

Passing

```
import static org.junit.Assert.*;
import org.junit.*;

public class BowlingTests {

    private Game game

    @Before
    public void setup() {
        game = new Game();
    }

    public void scoreForOneFrameGame() { ... }
    public void scoreForTwoFrameGame() { ... }

    @Test
    public void scoreForFirstFrameOfTwoFrames() {
        game.bowl(5, 3);
        game.bowl(4, 2);

        assertEquals(8, game.scoreForFrame(1));
    }
}
```

JUnit filters run before / after each test method.

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.*;

public class BowlingTests {

    private Game game

    public void setup() { ... }

    public void scoreForOneFrameGame() { ... }
    public void scoreForTwoFrameGame() { ... }
    public void scoreForFirstFrameOfTwoFrames() { ... }

    @Test
    public void scoreForSecondFrameOfTwoFrames() {
        game.bowl(5, 3);
        game.bowl(4, 2);

        assertEquals(8 + 6, game.scoreForFrame(2));
    }
}
```

Failure: Expected **14**, but got **8**.

Scoring Code

- Storing the total isn't enough any more.
- We need a quick design session.

1	2	3	4	5	6	7	8	9	10
5 3 8	4 2 14								

- How about an array to store the scores?

Scoring Code

Failing

First, we introduce the array and a pointer.

Next, we store each score in the array.

```
public class Game {

    private int total = 0;
    private int[] scores = new int
    private int nextThrowIndex

    public void bowl(int firstThro
        total += firstThrow + secondThrow;
        scores[nextThrowIndex++] = firstThrow;
        scores[nextThrowIndex++] = secondThrow;
    }

    public int score() { return total; }

    public int scoreForFrame(int frameNumber) {
        return 8;
    }
}
```

Scoring Code

Failing

```
public class Game {

    private int total = 0;
    private int[] scores = new int[21];
    private int nextThrowIndex = 0;

    public void bowl(int firstThrow, int secondThrow) {
        total += firstThrow + secondThrow;
        scores[nextThrowIndex++] = firstThr
        scores[nextThrowIndex++]
    }

    public int score() { retu

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            score += scores[scoreIndex++] + scores[scoreIndex++];
        }

        return score;
    }
```

And now use the array to calculate the score.

Scoring Code

Passing

```
public class Game {

    private int total = 0;
    private int[] scores = new int[21];
    private int nextThrowIndex = 0;

    public void bowl(int firstThrow, int secondThrow) {
        total += firstThrow + secondThrow;
        scores[nextThrowIndex++] = firstThrow;
        scores[nextThrowIndex++] = secondThrow;
    }

    public int score() { return total; }

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            score += scores[scoreIndex++] + scores[scoreIndex++];
        }

        return score;
    }
}
```

Scoring Code

Passing

```
public class Game {

    // We'll focus only on scoreForFrame from now on
    // the other methods and member variables remain the same

    public int scoreForFrame(int theFrame) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < theFrame; currFrame++) {
            score += scores[scoreIndex++] + scores[scoreIndex++];
        }

        return score;
    }
}
```

This is hard to read:
duplication and lots of +s

Next step: Look for code smells.

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex++];
            int secondThrow = scores[scoreIndex++];

            score += firstThrow + secondThrow;
        }

        return score;
    }
}
```

Refactor: Name complicated statements.

Next step: Another test.

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.*;

public class BowlingTests {

    private Game game

    public void setup() { ... }
    public void scoreForOneFrameGame() { ... }
    public void scoreForTwoFrameGame() { ... }
    public void scoreForFirstFrameOfTwoFrames() { ... }
    public void scoreForSecondFrameOfTwoFrames() { ... }

    @Test
    public void scoreForASpare() {
        game.bowl(6, 4);
        game.bowl(5, 1);

        assertEquals(15, game.scoreForFrame(1));
    }
}
```

Scoring Code

Failing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex++];
            int secondThrow = scores[scoreIndex++];

            score += firstThrow + secondThrow;
        }

        return score;
    }
}
```

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex++];
            int secondThrow = scores[sc
                                                    ];
            score += firstThrow + s

            if (firstThrow + secon                  == 10) {
                score += scores[scoreIndex];
            }
        }

        return score;
    }
}
```

Should this be scoreIndex or scoreIndex + 1?

Pass: Expected **15**, got **15**.

Next step: Look for code smells.

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(
        int score = 0, scoreI

        for (int currFrame =
            int firstThrow  = scores[scoreIndex++];
            int secondThrow = scores[scoreIndex++];

            score += firstThrow + secondThrow;

            if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex];
            }
        }

        return score;
    }
}
```

Incrementing
of scoreIndex is complicating this
method.

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex];
            int secondThrow = scores[scoreIndex+1];

            score += firstThrow + secondThrow;

            if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex+2];
            }

            scoreIndex += 2;
        }

        return score;
    }
}
```

Test Case

Failing

```
import static org.junit.Assert.*;
import org.junit.*;

public class BowlingTests {

    private Game game

    public void setup() { ... }
    public void scoreForOneFrameGame() { ... }
    public void scoreForTwoFrameGame() { ... }
    public void scoreForFirstFrameOfTwoFrames() { ... }
    public void scoreForSecondFrameOfTwoFrames() { ... }
    public void scoreForASpare() { ... }

    @Test
    public void scoreForAStrike() {
        game.bowl(10, 0);
        game.bowl(5, 1);

        assertEquals(16, game.scoreForFrame(1));
    }
}
```

Scoring Code

Failing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex];
            int secondThrow = scores[scoreIndex+1];

            score += firstThrow + secondThrow;

            if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex+2];
            }

            scoreIndex += 2;
        }

        return score;
    }
}
```

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex];
            int secondThrow = scores[scoreIndex+1];

            score += firstThrow + secondThrow;

            if (firstThrow == 10) {
                score += scores[scoreIndex+2] + scores[scoreIndex+3];
            } else if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex+2];
            }

            scoreIndex += 2;
        }

        return score;
    }
}
```

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame
            int firstThrow
            int secondThrow =
            score += firstThrow + secondThrow;

            if (firstThrow == 10) {
                score += scores[scoreIndex+2] + scores[scoreIndex+3];
            } else if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex+2];
            }

            scoreIndex += 2;
        }

        return score;
    }
}
```

Let's merge this into the if statement.

Refactor: Break up long method.

Scoring Code

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0, scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            int firstThrow  = scores[scoreIndex];
            int secondThrow = scores[scoreIndex+1];

            if (firstThrow == 10) {
                score += scores[scoreIndex+2] + scores[scoreIndex+3];
            } else if (firstThrow + secondThrow == 10) {
                score += scores[scoreIndex+2];
            } else {
                score += firstThrow + secondThrow;
            }

            scoreIndex += 2;
        }

        return score;
    }
}
```

Refactor: Break up long method.

Scoring Code

Passing

```
public class Game {

    public int scoreForFrame(int frameNumber) {
        int score = 0;
        scoreIndex = 0;

        for (int currFrame = 0; currFrame < frameNumber; currFrame++) {
            if (isStrike()) {
                score += 10 + nextTwoBalls();
            } else if (isSpare()) {
                score += 10 + nextBall();

            } else {
                score += ballsInThisFrame();
            }

            nextFrame();
        }

        return score;
    }
}
```

A challenge

TDD a Game of Life

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with < 2 live neighbours dies (underpopulation)
- Any live cell with > 3 live neighbours dies (overcrowding)
- Any live cell with 2 or 3 live neighbours lives on
- Any dead cell with 3 live neighbours becomes a live cell

The initial pattern is randomly generated.

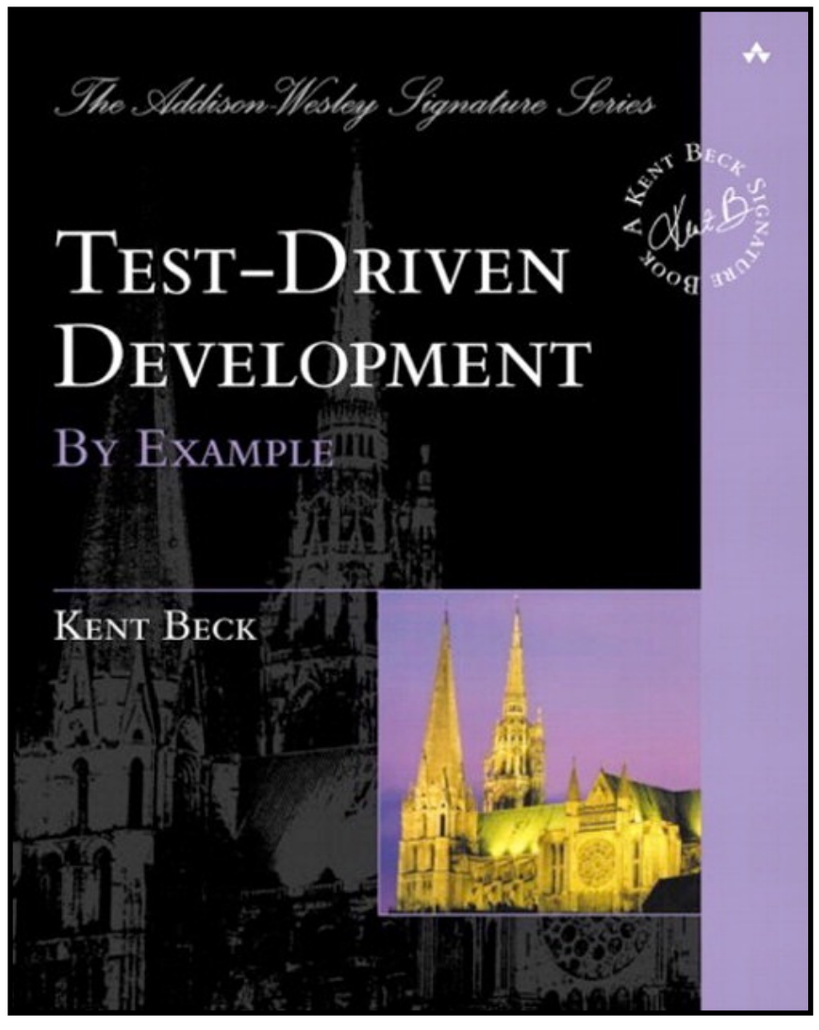
http://coderetreat.org/gol

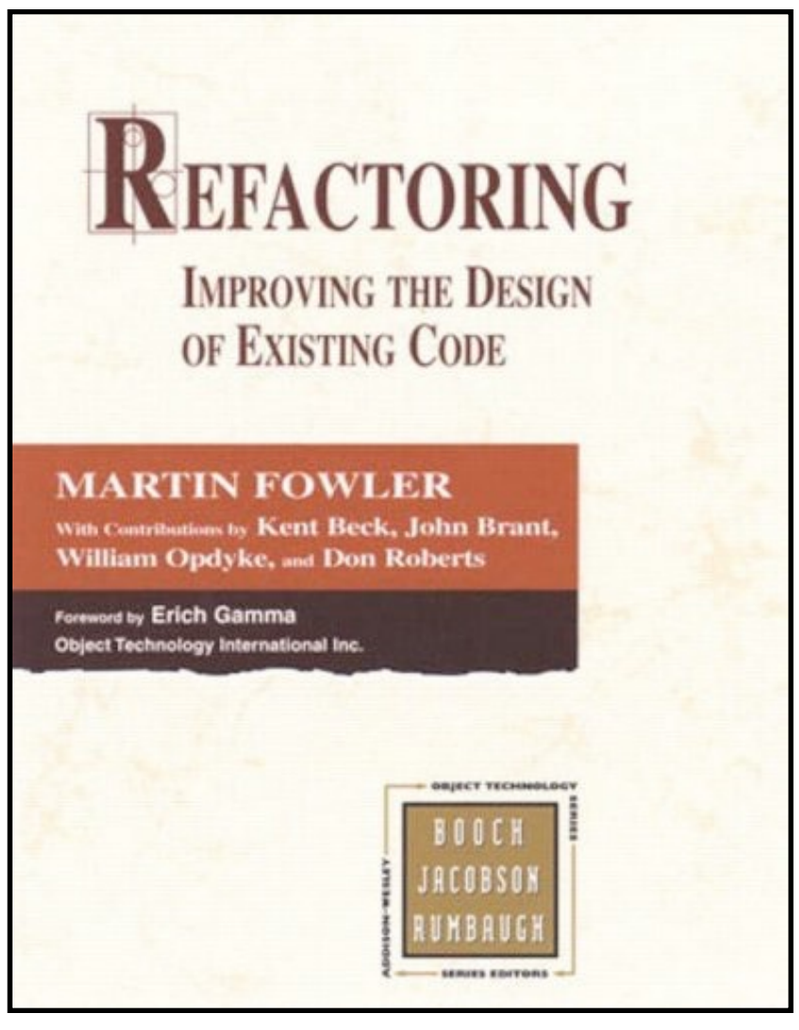
And finally...

Strategies for testing large systems:

- Dependency injection
- Test Doubles
- Mocks, Spies, Stubs, Dummies & Fakes
  http://www.martinfowler.com/bliki/TestDouble.html

TDD Resources





Conclusions

- TDD provides:
  - a simple process that is easy to follow
  - a suite of automated tests that can be used to check the correctness of code after we have to change it
  - a program that is unlikely to have redundant parts and that has been (and can continue to be) refactored