Test-Driven Development and Refactoring

Software Design and Analysis

CSCI 2040

Objectives

- Introduce two important development practices in the context of the case studies:
  - Test-Driven Development
  - Refactoring

Motivation

"Logic is the art of going wrong with confidence"

--Joseph Wood Krutch

Introduction

- Extreme Programming (XP) promoted an important testing practice: writing the tests *first*.
- It also promoted continuously refactoring code to improve its
  - qualityless duplication,
  - increased clarity, and so forth.
- Modern tools support both practices,
  - and many OO developers swear by their value.

Test-Driven Development

- An excellent practice promoted by the iterative and agile XP method, applicable to the UP (as most XP practices are), is **test-driven development** (TDD).
- This introduction will focus on TDD application to **unit testing individual classes**.

Test Code is Written First

- In OO unit testing TDD-style, test code is written *before* the class to be tested,
  - and the developer writes unit testing code for nearly *all* production code.
- The basic rhythm is to write a little test code, then write a little production code,
  - make it pass the test,
  - then write some more test code, and so forth.
- ***Key Point***: The test is written *first*, imagining the code to be tested is written.

Test-Driven Development Advantages

Test-Driven Development Advantages

- The unit tests actually get written..
- Programmer satisfaction leading to more consistent test writing.
- Clarification of detailed interface and behavior
- Provable, repeatable, automated verification
- The confidence to change things..

Unit Test Actually Get Written

- With TDD the Unit Test Actually Get Written
  - Human (or at least programmer) nature is such that avoidance of writing unit tests is very common,
    - Especially, if left as an afterthought.

Programmer Satisfaction Leading to More Consistent Test Writing

- This is more important than it sounds.. for sustainable, enjoyable testing work.
- If, following the traditional style, a developer first writes the production code, informally debugs it, and then as an afterthought is expected to add unit tests, it does not feel satisfying..
  - **This is *test-last development*, also known as *Just-this-one-time-I'll-skip-writing-the-test* development.**
  - It is human psychology!

Programmer Satisfaction Leading to More Consistent Test Writing

- If the test is written first, we feel a worthwhile challenge
  - and question in front of us: Can I write code to pass this test?
- And then, after the code is cut to pass the tests, there is some feeling of accomplishment meeting a goal.
  - And a very useful goal an executable, repeatable test.
- The psychological aspects of development cannot be ignored programming is a human endeavor..

Clarification of Detailed Interface & Behavior

- This sounds subtle, but it turns out in practice to be a major value of TDD!
- As you write the test code, you must *imagine* that the object code exists.
  - For example, if in your test code you write *sale.makeLineItem(description, 3)* to test the *makeLineItem* method (which doesn't exist yet)
  - you must think through the details of the public view of the method its name, return value, parameters, and behavior.
- That reflection improves or clarifies the detailed design.

Provable, Repeatable, Automated Verification

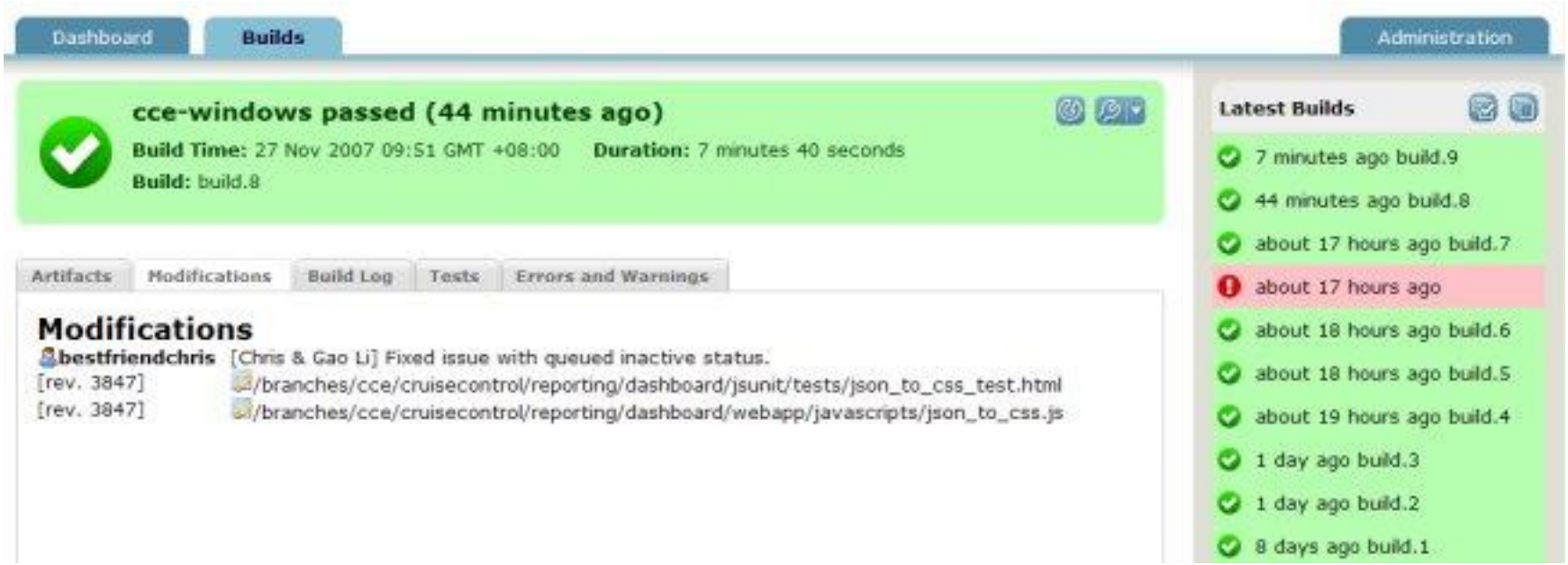
- Obviously, having hundreds or thousands of unit tests that build up over the weeks provides some meaningful verification of correctness.
  - And because they can be run automatically, it is easy.
  - Over time, as the test base builds from 10 tests to 50 tests to 500 tests, the early, more painful investment in writing tests starts to really feel like it is paying off as the size of the application grows.

The Confidence to Change Things

- In TDD, there will eventually be hundreds or thousands of unit tests (a unit test class for each production class).
- When a developer needs to change existing code written by themselves or others..
  - there is a unit test suite that can be run, providing immediate feedback if the change caused an error.
  - A popular free open source tool to automatically re-build the application and run all unit tests is **CruiseControl**.

CruiseControl

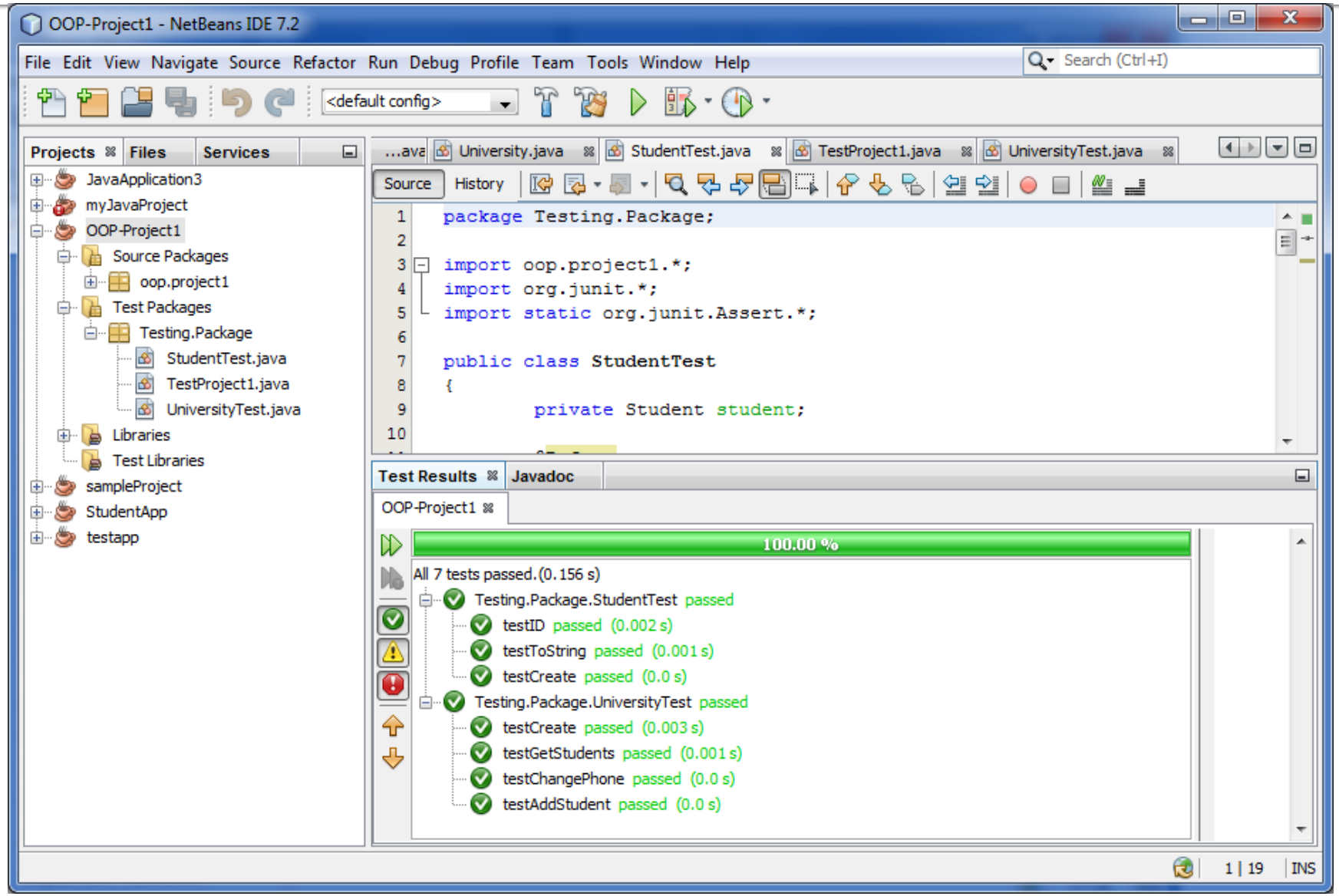
http://cruisecontrol.sourceforge.net/



TDD Tools

- The most popular unit testing framework is the **xUnit** family (for many languages), available at www.xunit.org.
- For Java, the popular version is **JUnit**. There's also an **NUnit** for .NET, and so forth.
  - JUnit is integrated into most of the popular Java IDEs, such as **Eclipse.**
- The xUnit family, and JUnit, was started by Kent Beck (creator of XP) and Eric Gamma (one of the Gang-of-Four design pattern authors, and the chief architect of the popular Eclipse IDE).

JUnit


OOP-Project1 - NetBeans IDE 7.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects
Files
Services
JavaApplication3
myJavaProject
OOP-Project1
Source Packages
oop.project1
Test Packages
Testing.Package
StudentTest.java
TestProject1.java
UniversityTest.java
Libraries
Test Libraries
sampleProject
StudentApp
testapp
University.java
StudentTest.java
TestProject1.java
UniversityTest.java
Source
History
1 package Testing.Package;
2
3 import oop.project1.*;
4 import org.junit.*;
5 import static org.junit.Assert.*;
6
7 public class StudentTest
8 {
9 private Student student;
10
Test Results
Javadoc
OOP-Project1
100.00 %
All 7 tests passed. (0.156 s)
Testing.Package.StudentTest passed
testID passed (0.002 s)
testToString passed (0.001 s)
testCreate passed (0.0 s)
Testing.Package.UniversityTest passed
testCreate passed (0.003 s)
testGetStudents passed (0.001 s)
testChangePhone passed (0.0 s)
testAddStudent passed (0.0 s)
1 | 19
INS

JUnit Example

- Suppose we are using JUnit and TDD to create the *Sale* class.
- *Before* programming the *Sale* class, we write a unit testing method in a *SaleTest* class that does the following:
  1. Create a *Sale* the thing to be tested (also known as the **fixture**).
  2. Add some line items to it with the *makeLineItem* method (the *makeLineItem* method is the public method we wish to test).
  3. Ask for the total, and verify that it is the expected value, using the *assertTrue* method. JUnit will indicate a failure if any *assertTrue* statement does not evaluate to *true*.

Pattern

- Each testing method follows this pattern:
  1. **Create the fixture.**
  2. **Do something to it (some operation that you want to test).**
  3. **Evaluate that the results are as expected.**

Key Points

- A ***key point*** to note is that we do *not* write all the unit tests for *Sale* first;
  - rather, we write only one test method, implement the solution in class *Sale* to make it pass, and then repeat.
- To use JUnit, you must create a test class that extends the JUnit *TestCase* class;
  - your test class inherits various unit testing behaviors.
- In JUnit you create a separate testing method for *each* Sale method that you want to test.
  - In general you will write unit testing methods (perhaps several) for each *public* method of the *Sale* class.
  - Exceptions include trivial (and usually auto-generated) *get* and *set* methods.

STEP 1: Create the Fixture

```
public class SaleTest extends TestCase
{
    // …

   // test the Sale.makeLineItem method
  public void testMakeLineItem()
  {
      // STEP 1: CREATE THE FIXTURE
      // -this is the object to test
      // -it is an idiom to name it 'fixture'
      // -it is often defined as an instance field rather than
      // a local variable
    Sale fixture = new Sale();

       //  set up supporting objects for the test
    Money total = new Money( 7.5 );
    Money price = new Money( 2.5 );
    ItemID id = new ItemID( 1 );
    ProductDescription desc =
             new ProductDescription( id, price, "product 1" );
```

STEP 2: Execute the Method to Test

```
    // STEP 2: EXECUTE THE METHOD TO TEST

    // NOTE: We write this code **imagining** there
    // is a makeLineItem method. This act of imagination
    // as we write the test tends to improve or clarify
    // our understanding of the detailed interface to
    // to the object. Thus TDD has the side-benefit of
    // clarifying the detailed object design.

    // test makeLineItem
sale.makeLineItem( desc, 1 );
sale.makeLineItem( desc, 2 );
```

STEP3: Evaluate the Results

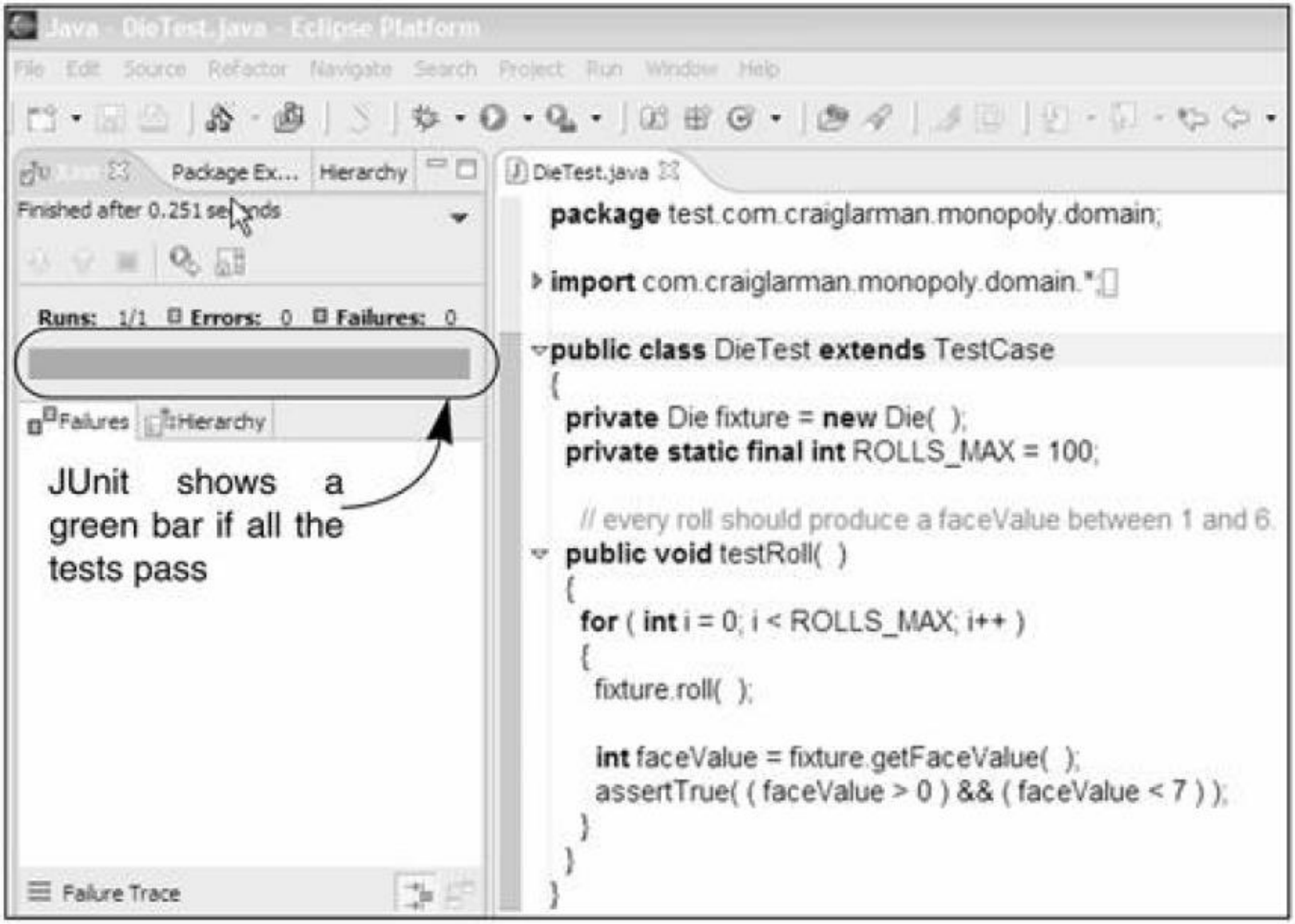
```
        // STEP 3: EVALUATE THE RESULTS

        // there could be many assertTrue statements
        // for a complex evaluation

        // verify the total is 7.5
      assertTrue( sale.getTotal().equals( total ));
    }
}
```

- Only after this *testMakeLineItem* test method is written do we then write the *Sale.makeLineItem* method to pass this test.
  - Hence, the term test-*driven* or test-*first* development.

IDE Support for TDD and xUnit

- Most IDEs have built-in support for some xUnit tool.
  - For example, Eclipse supports JUnit.
  - Junit includes a visual cue if all the tests pass when executed, it displays a green bar.
- This gave rise to the TDD mantra:
  - ***Keep the bar green to keep the code clean.***

Support for TDD and Junit in a Popular IDE, Eclipse



Refactoring

- **Refactoring** is a method to rewrite or restructure existing code without changing its external behavior,
  - applying small transformation steps combined with re-executing tests each step.
- Continuously refactoring code is another XP practice and applicable to all iterative methods (including the UP).
  - Ralph Johnson (one of the Gang-of-Four design pattern authors) and Bill Opdyke first discussed refactoring in 1990.
  - Beck (XP creator), along with Martin Fowler, are two other refactoring pioneers.

Essence of Refactoring

- The essence of refactoring is applying small *behavior preserving transformations* (each called a 'refactoring'), one at a time.
  - After each transformation, the unit tests are re-executed to prove that the refactoring did not cause a regression (failure).
  - Therefore, there's a relationship between refactoring and TDD all those unit tests support the refactoring process.
- Each refactoring is small, but a series of transformation followed by executing the unit tests again can produce a major restructuring of the code and design (for the better),
  - all the while ensuring the behavior remains the same.

What are the Activities and Goals of Refactoring?

- remove duplicate code
- improve clarity
- make long methods shorter
- remove the use of hard-coded literal constants

Code Smells..

- Code that's been well-refactored is short, tight, clear, and without duplication
  - it looks like the work of a master programmer..
- Code that doesn't have these qualities *smells bad* or has *code smells*.
- **Code smells** is a metaphor in refactoring, they are *hints* that something *may* be wrong in the code.
  - It might turn out to be alright and not need improvement.
  - **Code stench** truly putrid code crying out for clean up!

Code Smells..

- Some code smells include:
  - duplicated code
  - big method
  - Class with many instance variables
  - class with lots of code
  - strikingly similar subclasses
  - little or no use of interfaces in the design
  - high coupling between many objects

Remedy

- The remedy to smelly code are the refactorings.
- Like patterns, refactorings have names, such as *Extract Method.*
  - There are about 100 named refactorings;

Refactorings

- Here's a sample to get a sense of them:

Refactoring	Description
Extract Method	Transform a long method into a shorter one by factoring out a portion into a private helper method.
Extract Constant	Replace a literal constant with a constant variable.
Introduce Explaining Variable (specialization of Extract Local Variable)	Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.
Replace Constructor Call with Factory Method	In Java, for example, replace using the *new* operator and constructor call with invoking a helper method that creates the object (hiding the details).

The takeTurn method Before Refactoring

```
public class Player
{
   private Piece  piece;
   private Board  board;
   private Die[]  dice;
   // …

public void takeTurn()
{
       // roll dice
   int rollTotal = 0;
   for (int i = 0; i < dice.length; i++)
   {
      dice[i].roll();

       rollTotal += dice[i].getFaceValue();
    }

   Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
   piece.setLocation(newLoc);

}

} // end of class
```

The Code after Refactoring with Extract Method

```
public class Player
{
   private Piece  piece;
   private Board  board;
   private Die[]  dice;
   // …

public void takeTurn()
{
        // the refactored helper method
   int rollTotal = rollDice();

   Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
   piece.setLocation(newLoc);
}

private int rollDice()
{
   int rollTotal = 0;
   for (int i = 0; i < dice.length; i++)
   {
      dice[i].roll();
      rollTotal += dice[i].getFaceValue();
   }
   return rollTotal;
}

} // end of class
```

Extract Method Refactoring

- Notice in the listing that the *takeTurn* method in the *Player* class has an initial section of code that rolls the dice and calculates the total in a loop.
- This code is itself a distinct, cohesive unit of behavior;
  - we can make the *takeTurn* method shorter, clearer, and better supporting High Cohesion by extracting that code into a private helper method called *rollDice*.
- Notice that the *rollTotal* value is required in *takeTurn*, so this helper method must return the *rollTotal*.

Before Introducing an Explaining Variable

```
   // good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
   return( ( ( year % 400 ) == 0 ) ||
         ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
```

Introduce Explaining Variable

- One of favorite simple refactorings is *Introduce Explaining Variable*
  - it clarifies, simplifies, and reduces the need for comments

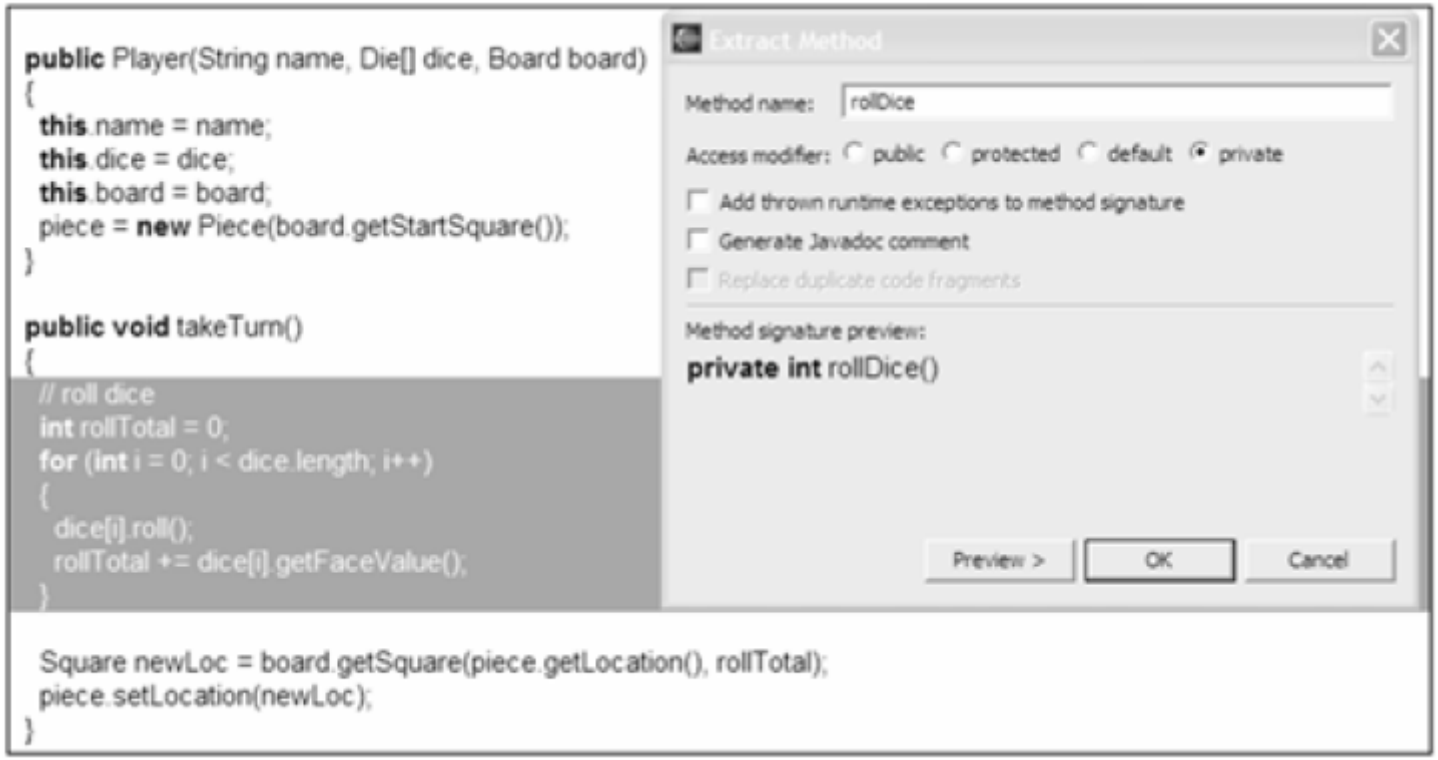
After Introducing an Explaining Variable

```
   // that's better!
boolean isLeapYear( int year )
{
   boolean isFourthYear = ( ( year % 4 ) == 0 );
   boolean isHundrethYear = ( ( year % 100 ) == 0);
   boolean is4HundrethYear = ( ( year % 400 ) == 0);
   return (
      is4HundrethYear
      || ( isFourthYear && ! isHundrethYear ) );
}
```

IDE Support for Refactoring

- Most of the dominant IDEs include automated refactoring support.
- For an example in the Eclipse IDE of applying the Extract Method refactoring.

IDE Before Refactoring



IDE After Refactoring

```
public void takeTurn()
{
  int rollTotal = rollDice();

  Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
  piece.setLocation(newLoc);
}

private int rollDice()
{
  // roll dice
  int rollTotal = 0;
  for (int i = 0; i < dice.length; i++)
  {
    dice[i].roll();
    rollTotal += dice[i].getFaceValue();
  }
  return rollTotal;
}
```

IDE After Refactoring

- The *rollDice* method is automatically generated, as is the call to it from the *takeTurn* method.
  - Notice that the tool is smart enough to see the need to return the *rollTotal* variable. Nice!

Recommended Recourses

- For TDD on the Web:
  - www.junit.org
  - www.testdriven.com
- For refactoring on the Web:
  - www.refactoring.com
  - www.c2.com/cgi/wiki?WhatIsRefactoring

  (a major Wiki on many subjects)

Quiz

- Describe test-driven development in XP.
- What does the metaphor “code smells” stands for?
- What are the example refactorings? Provide a description for each of them.
- Does the IDE support for refactoring?

Actions

- Review Slides.
- Read Chapter 21 (3rd edition)
  - *Test-Driven Development and Refactoring, Applying UML and Patterns*, Craig Larman