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 (TDD)
  - Refactoring

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 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

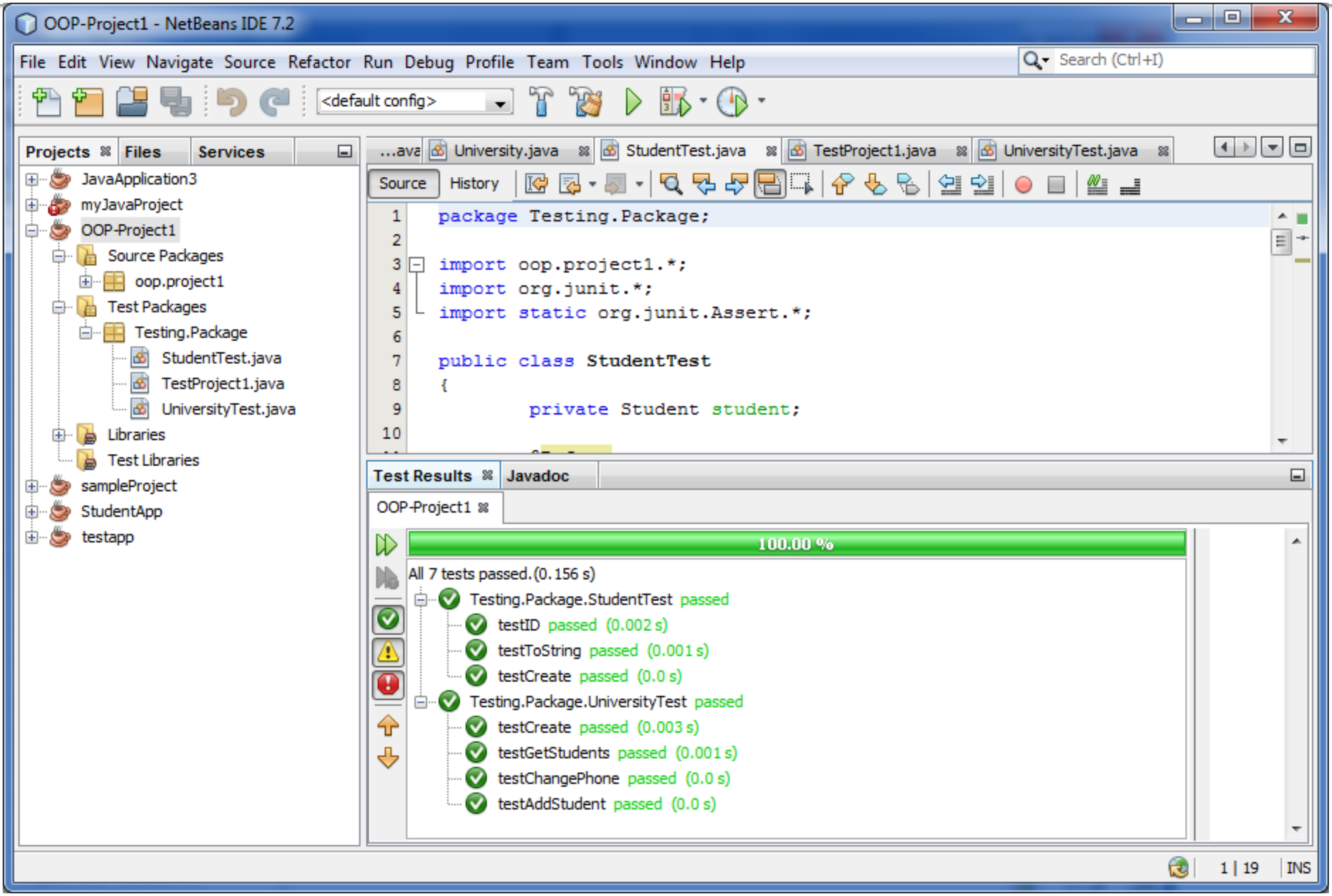
Test-Driven Development Advantages

- 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..

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



```
package Testing.Package;

import oop.project1.*;
import org.junit.*;
import static org.junit.Assert.*;

public class StudentTest
{
        private Student student;
```

```
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)
```

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.**

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

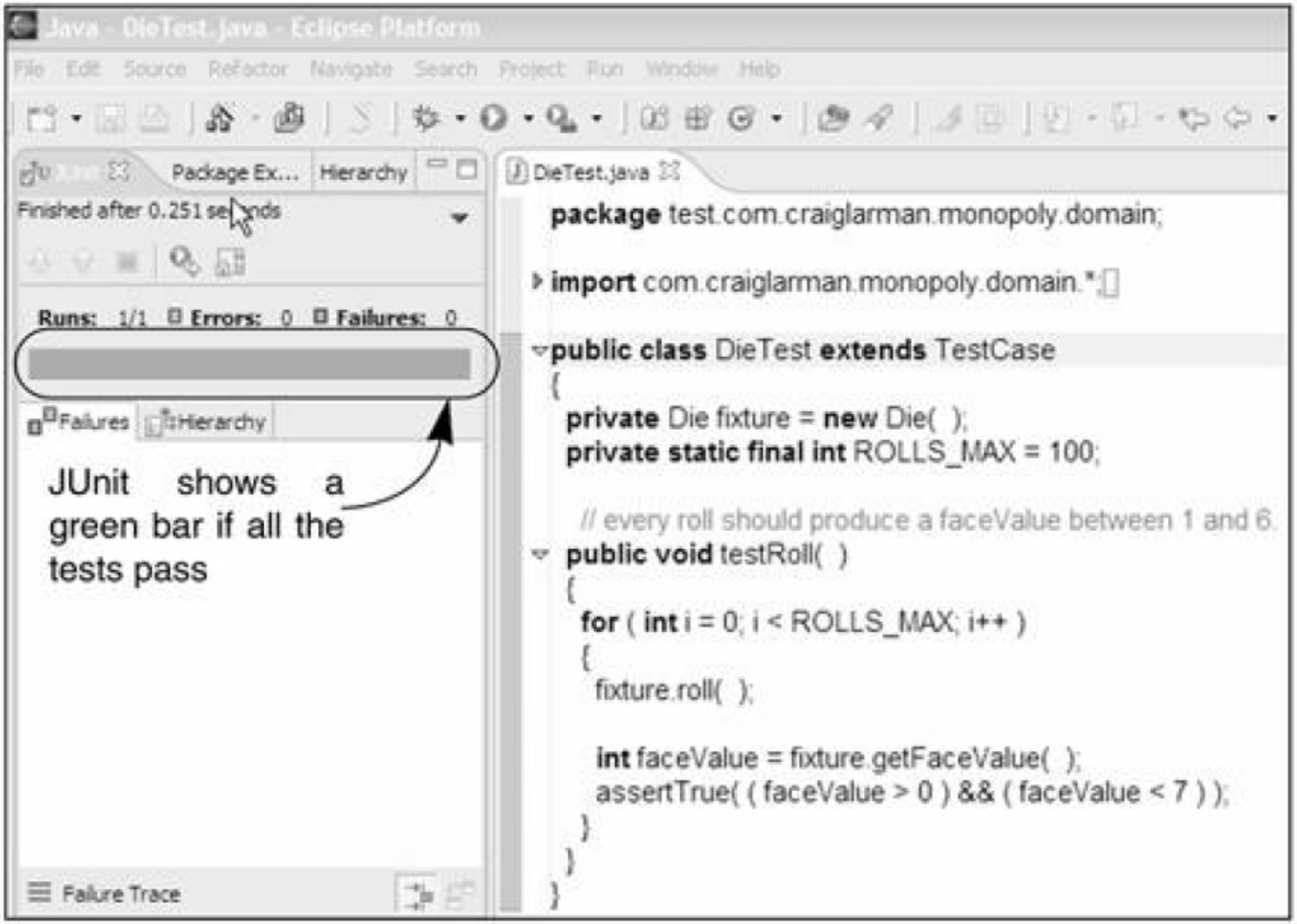
```
           // STEP 3: EVALUATE THE RESULTS

           // 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.

Support for TDD and Junit in a Popular IDE, Eclipse


Java - DieTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Ex... Hierarchy
DieTest.java
Finished after 0.251 seconds
Runs: 1/1 Errors: 0 Failures: 0
Failures Hierarchy
Failure Trace

JUnit shows a green bar if all the tests pass

```
package test.com.craiglarman.monopoly.domain;

import com.craiglarman.monopoly.domain.*;

public class DieTest extends TestCase
{
 private Die fixture = new Die( );
 private static final int ROLLS_MAX = 100;

  // every roll should produce a faceValue between 1 and 6.
 public void testRoll( )
 {
  for ( int i = 0; i < ROLLS_MAX; i++ )
  {
   fixture.roll( );

   int faceValue = fixture.getFaceValue( );
   assertTrue( ( faceValue > 0 ) && ( faceValue < 7 ) );
  }
 }
}
```

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.

What are the Activities and Goals of Refactoring?

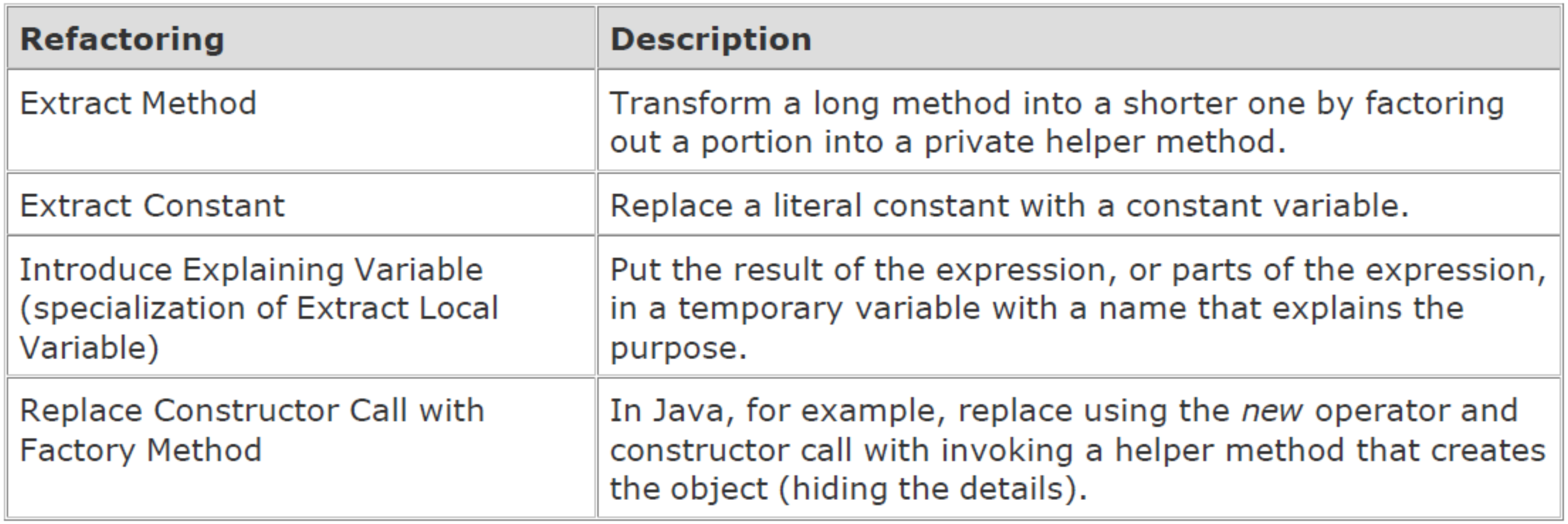
- remove duplicate code
- improve clarity
- make long methods shorter
- remove the use of hard-coded literal constants

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!

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
```

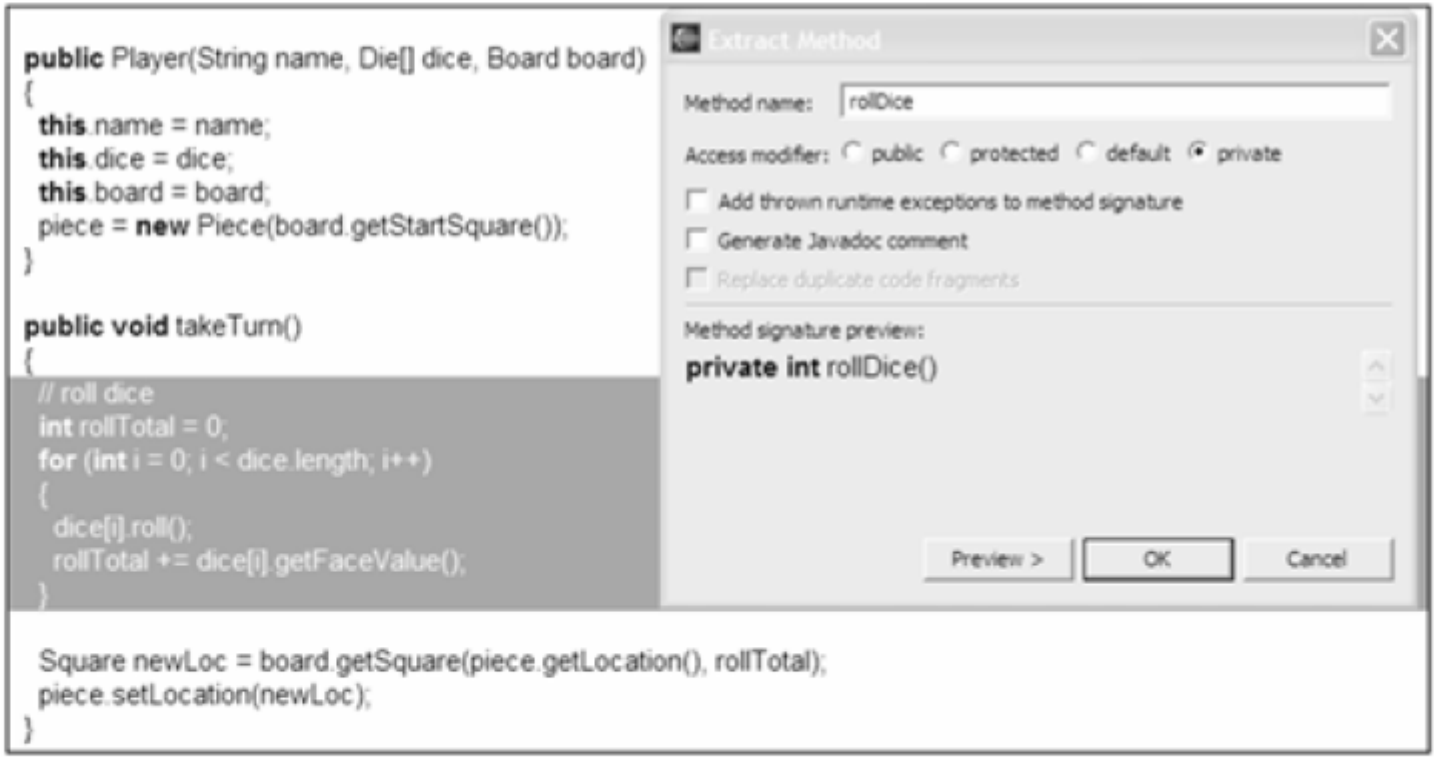
Before Introducing an Explaining Variable

```
   // 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 ) ) );
}
```

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 Before Refactoring



```
public Player(String name, Die[] dice, Board board)
{
  this.name = name;
  this.dice = dice;
  this.board = board;
  piece = new Piece(board.getStartSquare());
}

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);
}
```

Method name: rollDice

Access modifier: ( ) public ( ) protected ( ) default [✓ private]

[ ] Add thrown runtime exceptions to method signature

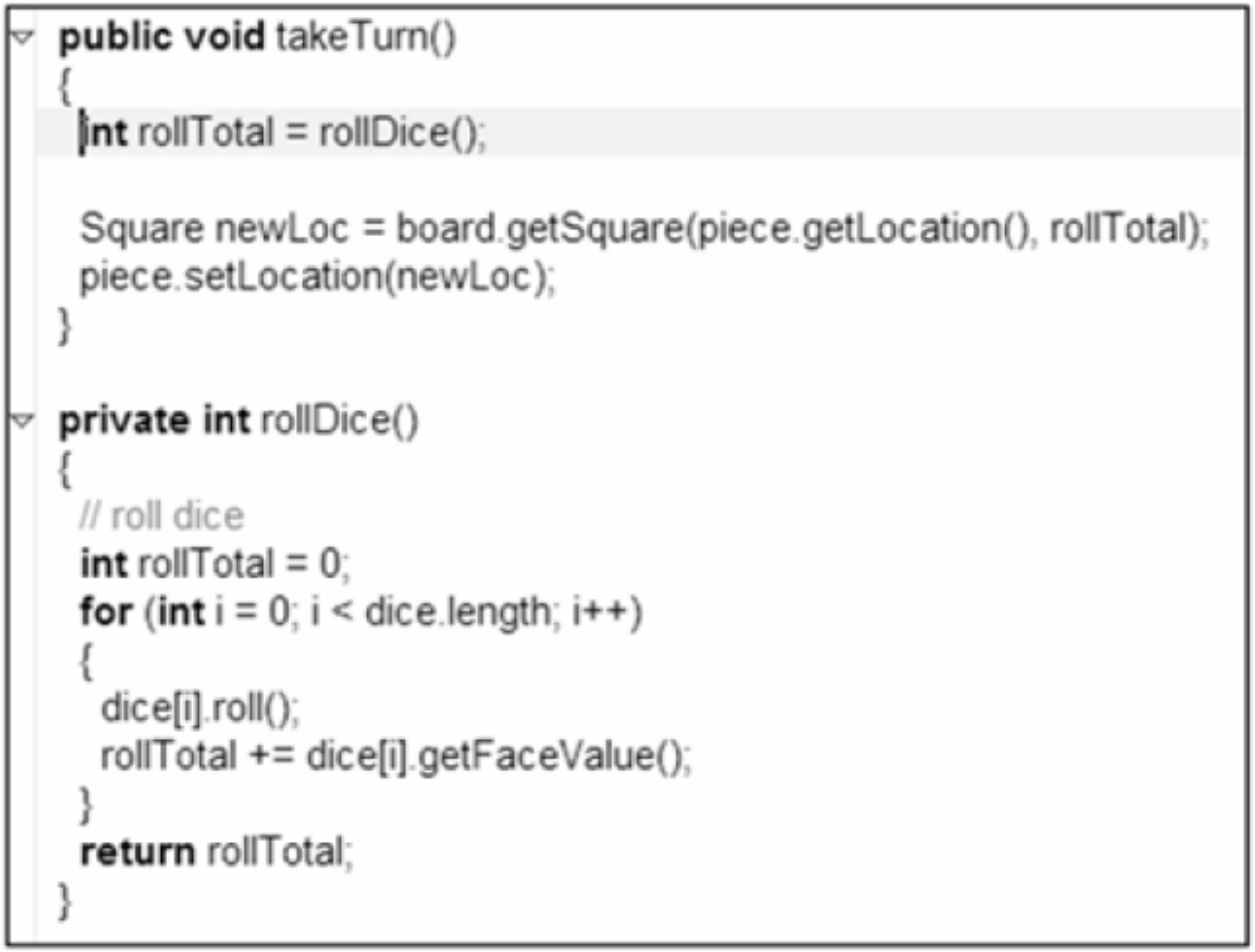
[ ] Generate Javadoc comment

[ ] Replace duplicate code fragments

Method signature preview:

```
private int rollDice()
```

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;
}
```

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