So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code
- **Test-Driven Development and Refactoring**

Part: 4 Elaboration Iteration 2—More Patterns

- GRASP: More Objects with Responsibilities
- Applying GoF Design Patterns

Test-Driven Development and Refactoring

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Logic is the art of going wrong with confidence.

—Joseph Wood Krutch

Intro

Extreme Programming (XP) **promotes**

- Writing the **tests** first
- Continuous code **refactoring**

Why?

- **Improve its quality**
- **Less duplication**
- **Increased clarity**

Modern tools **support** practices, OO developers **swear** by their value

Unit Testing First

Testing individual components, individual classes

In OO unit testing TDD-style (Test Driven Dev.), test code is written before the class to be tested

1. Imagining a production code,
2. Write a little test code,
3. Then write a little production code,
4. Make it pass the test,
5. … then 1 & write some more test code, etc.

Unit Testing First, Why?

1. Unit tests get written—

Human nature, if left as an afterthought, writing unit test is avoided

2. Programmer Satisfaction—

- Test-last, or Just-this-one-time-I'll-skip-writing-the-test development Traditional style,
  - developer writes production code, debugs, then add unit tests,
  - it doesn't feel satisfying, you may even **hate** it!

- Human psychology. Test is written first, Pass Test, Can you?, I challenge you or myself?
  - Code is cut to pass the tests, feel of accomplishment—meeting a goal!

Unit Testing First, Why?

3. Clarification of detailed interface and behavior—

Writing tests, you imagine code exists, details of public view of methods

- Name, return value, parameters, and behavior

That improves/clarifies the detailed design;

designing your code before writing it

4. Provable, repeatable, automated verification—

Having hundreds or thousands of unit tests provides verification of correctness, runs automatically, it's easy

Unit Testing First, Why?

5. Confidence in change—

Unit test suite provides immediate feedback if the change caused an error

You write your own tests for your own code—

Who is better than the authors to write unit tests of their own code?

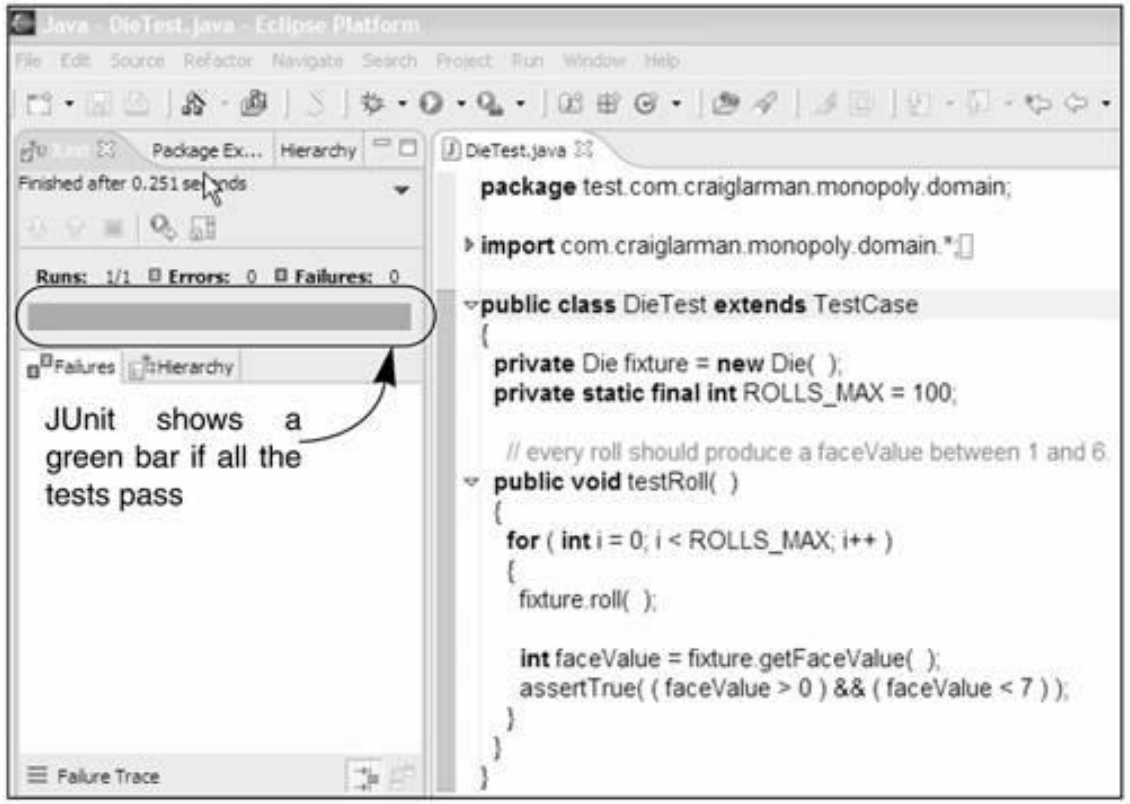
Frameworks

Most popular unit testing framework is the xUnit family (for many languages)

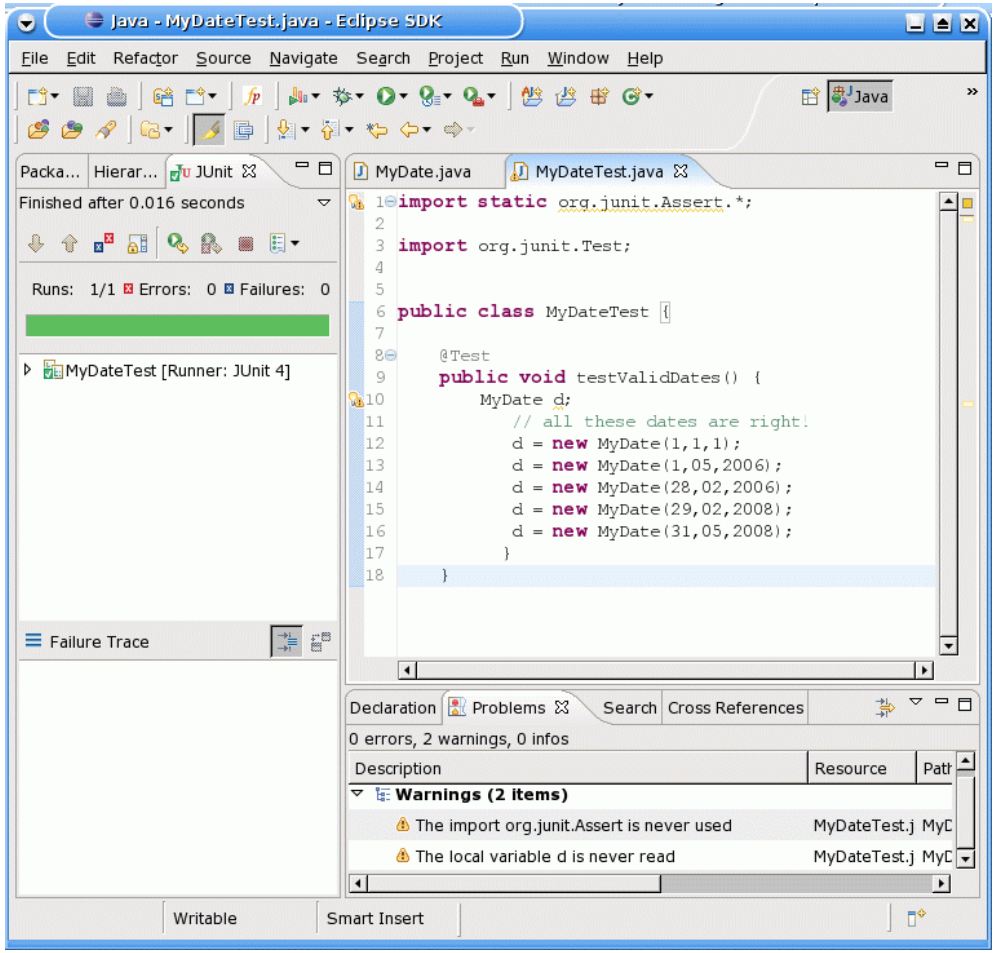
JUnit for Java, NUnit for .NET, etc. xUnits are integrated into IDEs (e.g. Eclipse, MS Visual Studio)

Keep the bar green to keep the code clean

TDD and JUnit in a popular IDE, Eclipse



TDD and JUnit in a popular IDE, Eclipse



TDD —POS

Before programming **Sale** class, write 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 for the public ***makeLineItem*** method to test

3. Ask for the total, and verify that it is the expected value, using the ***assertTrue*** method

TDD How —POS

Do not write all the unit tests for **Sale** first; rather,

- Write only one test method,
- Implement the solution in class **Sale** to make it pass,
- and then repeat

To use xUnit, create test class that extends xUnit **TestCase** class

Write unit testing methods (perhaps several) **for each public method** of the **Sale** class

TDD How —POS

Exceptions include trivial (and usually auto-generated) get and set methods

To test method ***MakeLineItem***, it is an idiom to name the testing method ***testMakeLineItem***

1. Write ***testMakeLineItem*** test method,
2. then ***Sale.makeLineItem*** method to pass test

SaleTest

```
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" );
```

SaleTest

```
        // 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 );


         // 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 ));
  }
}
```

Refactoring

Is a structured, disciplined method to rewrite or **restructure** existing code without changing its external behavior

Via applying **small transformation** steps combined with **re-executing tests** each step

An XP practice, part of iterative methods, including UP

Refactoring and TDD

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 failure

Relationship between refactoring and TDD—
Unit tests support refactoring

Refactoring, Why?

Each refactoring is small

A series of transformations—each followed by executing the unit tests again and again

Produces a major restructuring of code and design (for the better), while ensuring behavior remains the same

Code Smells

Code that's been well-refactored is short, tight, clear, and without duplication—A work of a master programmer. Code that doesn't have these qualities smells bad or has code smells, poor design

Signs of Code Smell

- Duplicated code
- Big method
- Class with many instance variables
- Class with lots of code, non cohesive
- Strikingly similar subclasses
- Little or no use of interfaces in the design
- High coupling between many objects

Refactoring Activities

- Remove duplicate code
- Improve clarity
- Make long methods shorter
- Remove the use of hard-coded literal constants

Refactorings Have Names, 100!

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	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	Replace using the new operator and constructor call with invoking a helper method that creates the object (hiding details)

Extract Method Refactoring Example

Player.*takeTurn* has an initial section of code that rolls the dice and calculates the total in a loop

Make the ***takeTurn*** method shorter, clearer, and better supporting High Cohesion by extracting that code into a private helper method called ***rollDice***

Extract Method Refactoring —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
```

Extract Method Refactoring —After Refactoring

```
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
```

Introduce Explaining Variable —Before Refactoring

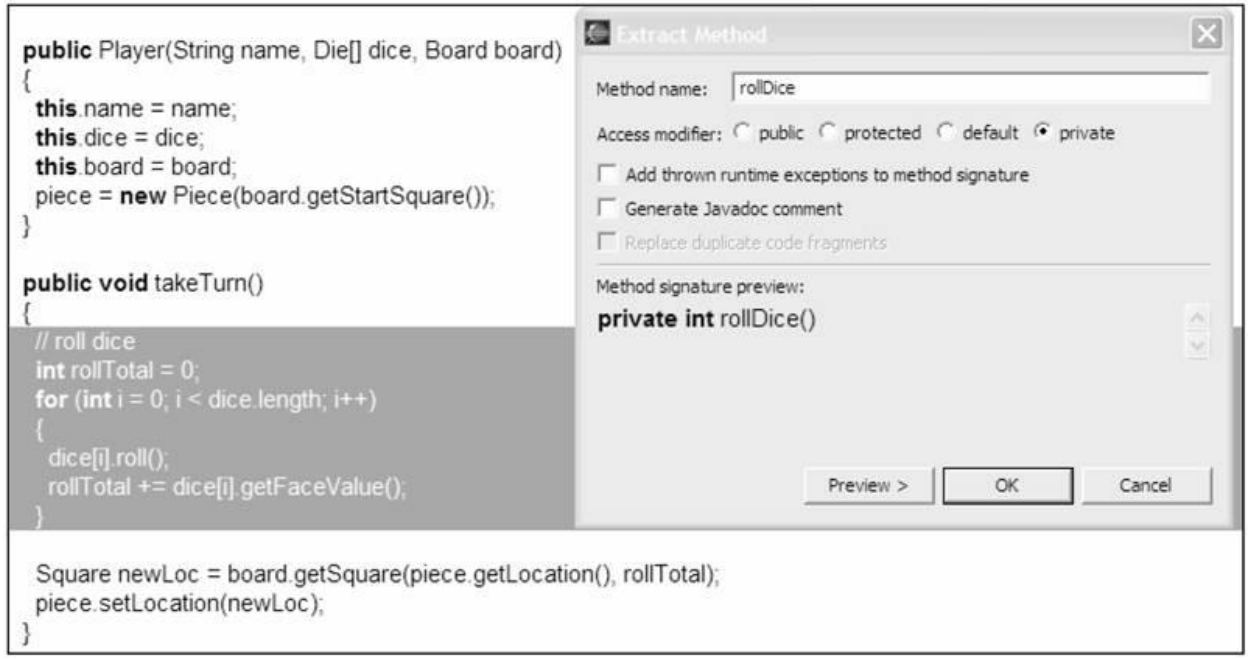
```
   // good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
   return( ( ( year % 400 ) == 0 ) ||
         ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
```

Introduce Explaining Variable —After Refactoring

Clarifies, simplifies, and reduces the need for comments

```
   // 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 —Before Refactoring



IDE Support for Refactoring —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;
}
```