THE UNIVERSITY OF MELBOURNE

SWEN30006 Software Modelling and Design

ON TO OBJECT DESIGN

Larman Chapter 14

I do not like this word 'bomb.' It is not a bomb.
It is a device that is exploding.

—Ambassador Jacques le Blanc on nuclear 'weapons'

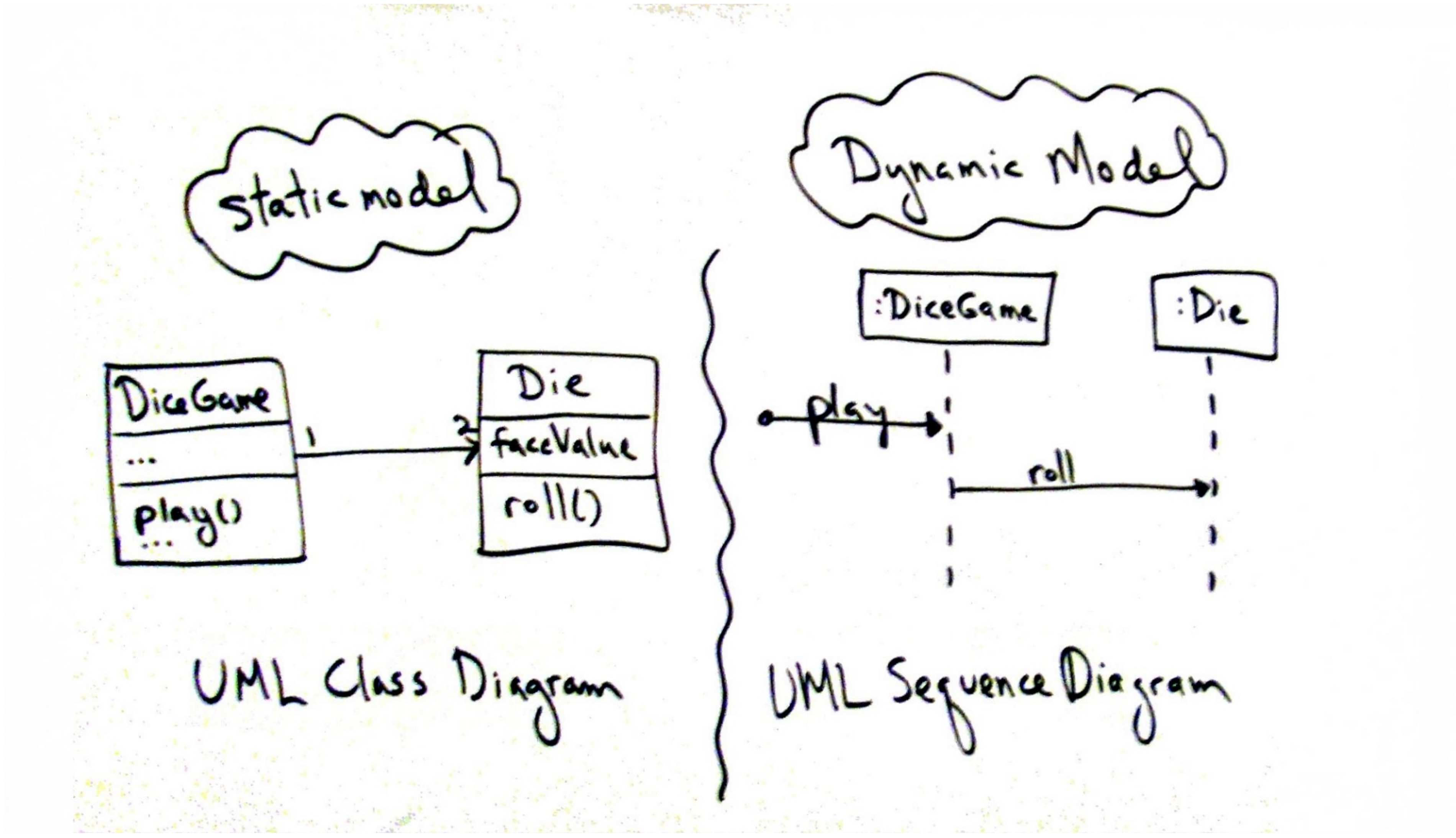
How do Developers Design Objects?

1. **Code.** From mental model to code. Design-while-coding (Java, C#, ...), ideally with an IDE (e.g., Eclipse or Visual Studio) which supports refactoring and other high-level operations.
2. **Draw, then code.** Drawing UML on a whiteboard or UML CASE tool (e.g. EA), then switching to **1.**
3. **Only draw.** The tool generates everything from diagrams! *Many a dead tool vendor has washed onto the shores of this steep island.* “Only draw” is a misnomer; it still involves a programming language attached to the graphic elements.

Agile Modelling & Lightweight UML Drawing

- Modelling with other developers
- Static and dynamic models
  - Class *and* interaction diagrams
  - Create several models in parallel
- Hand draw, and/or
  - White boards, large surface area, digital capture
- UML tool
  - IDE integration, reverse or round trip (class and interaction diagrams)
- How long?: few hours to a day near iteration start (3wk iter.)

Static and Dynamic UML Modelling



Object Design Skill vs UML Drawing Skill

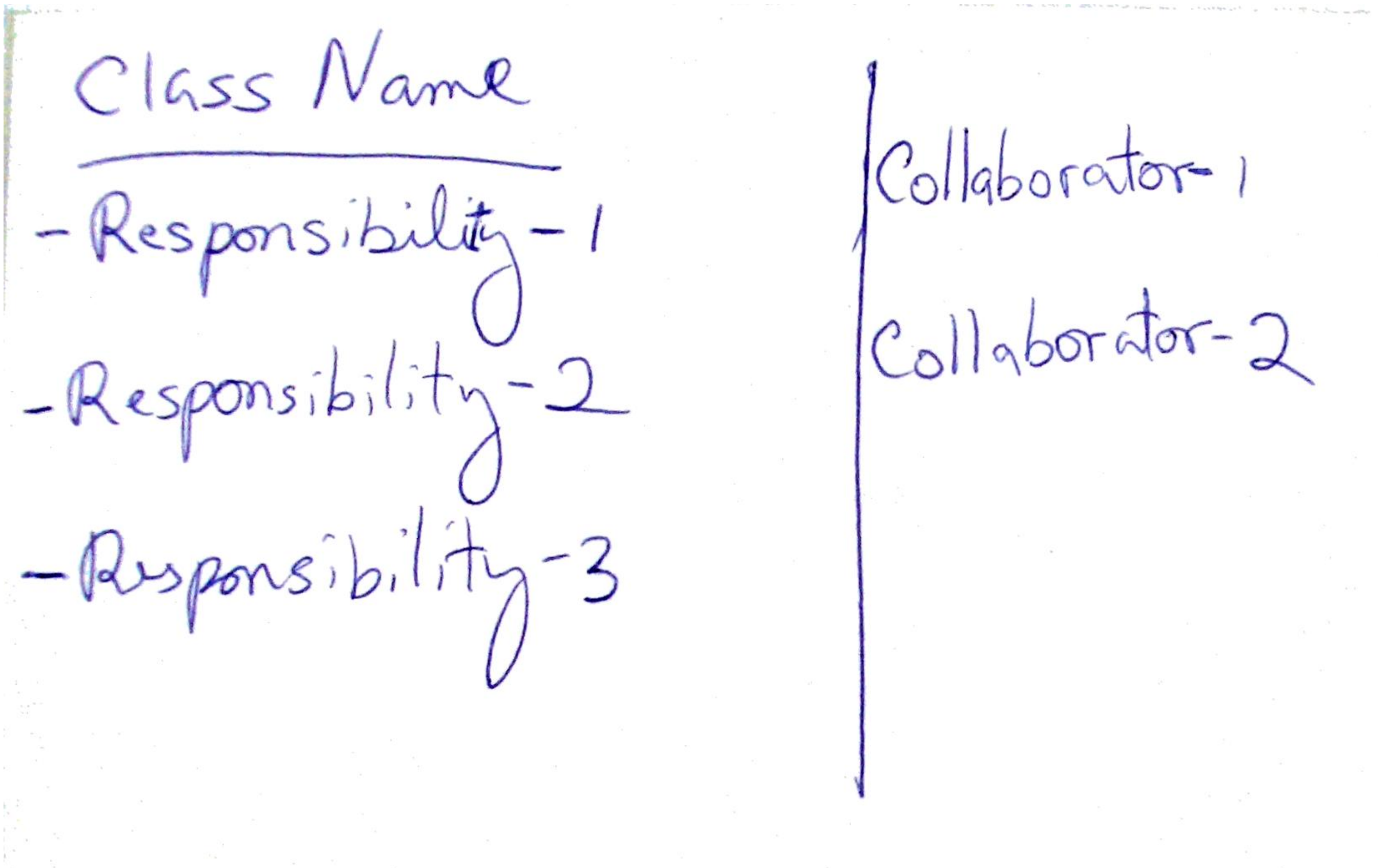
- UML models should reflect decision making about the design
- UML models should use correct notation as a principle of communication
- However, of greatest importance is object design skill, not UML drawing skill
- Object design requires knowledge of
  - principles of responsibility assignment
  - design patterns

Object Design: CRC Cards

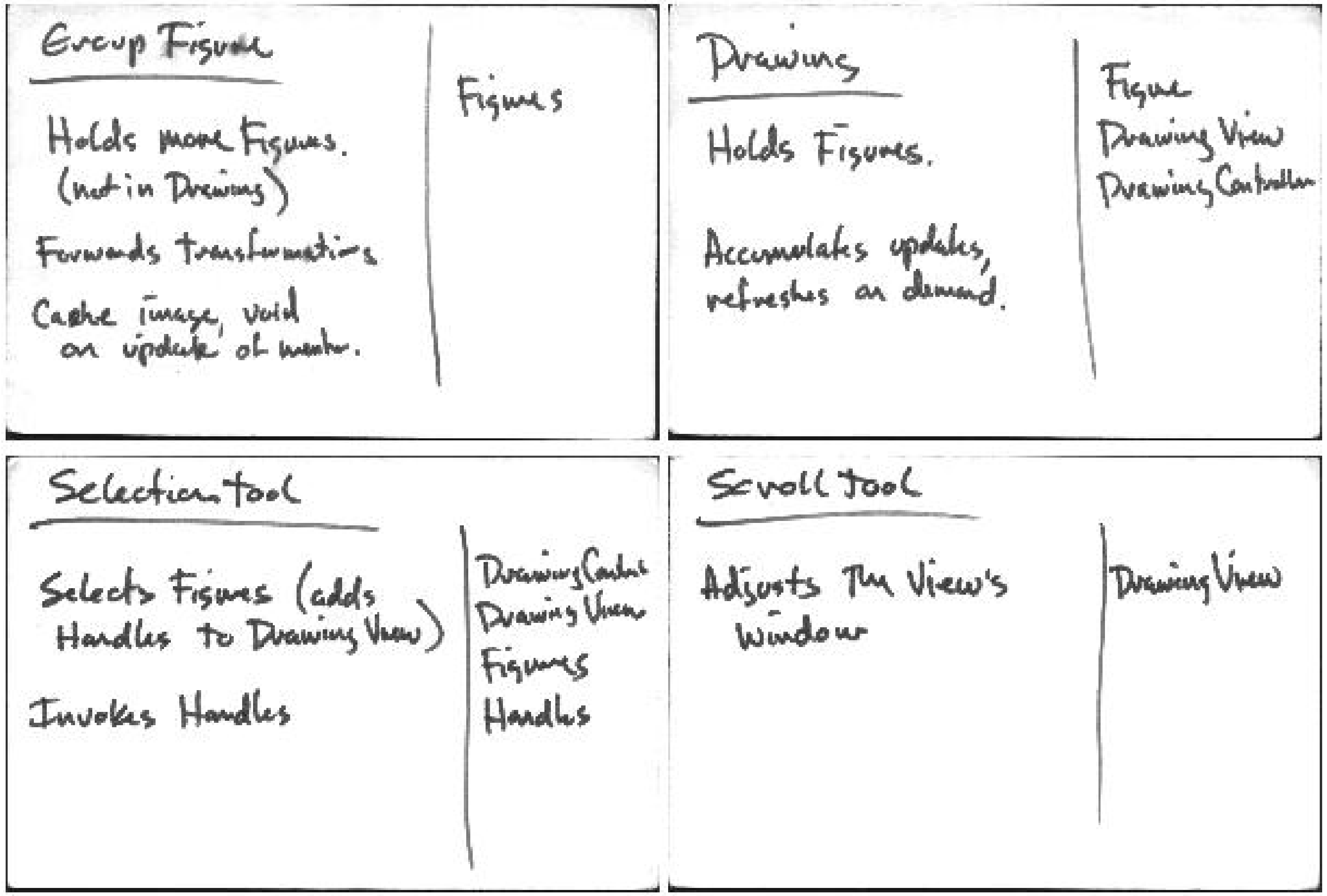
*Class Responsibility Collaboration (**CRC**) cards*

- are a popular text-oriented object design technique
- involves a group considering "what-if" scenarios
- based around index cards
  - one per class
  - work to list responsibilities and collaborators
- details of approach can vary (e.g. person holds class)
- can be a primary or supplementary approach

Template for a CRC Card



Typical Detail Level: 4 CRC Cards



Group Figure	
Holds more Figures. (not in Drawing)	Figures
Forwards transformations	
Cache image, void on update of member.	

Drawing	
Holds Figures.	Figure
Accumulates updates, refreshes on demand.	Drawing View
	Drawing Controller

Selection tool	
Selects Figures (adds Handles to Drawing View)	Drawing Controller
Invokes Handles	Drawing View
	Figures
	Handles

Scroll Tool	
Adjusts The View's window	Drawing View

SWEN30006
Software Modelling and Design

TEST-DRIVEN DEVELOPMENT & REFACTORING

Larman Chapter 21

Logic is the art of going wrong with confidence.

—Joseph Wood Krutch

Test-Driven Development

- Development practise in which test code is written before the code that it will test, e.g.
  - acceptance tests at start of iteration
  - unit tests before the corresponding class
- General approach:
  - alternate between test code & production code
  - ensure production code passes tests before proceeding
- Promoted in iterative and agile practice (esp. XP)

Advantages of Writing Tests First

- Tests actually get written
- Programmer satisfaction leading to more consistent test writing
- Clarification of detailed interface and behaviour
- Proven, repeatable, automated verification
  - Build up a suite of tests, easy to re-run
- The confidence to make changes
  - Tests can check for unwanted change, and can be changed to check for wanted change

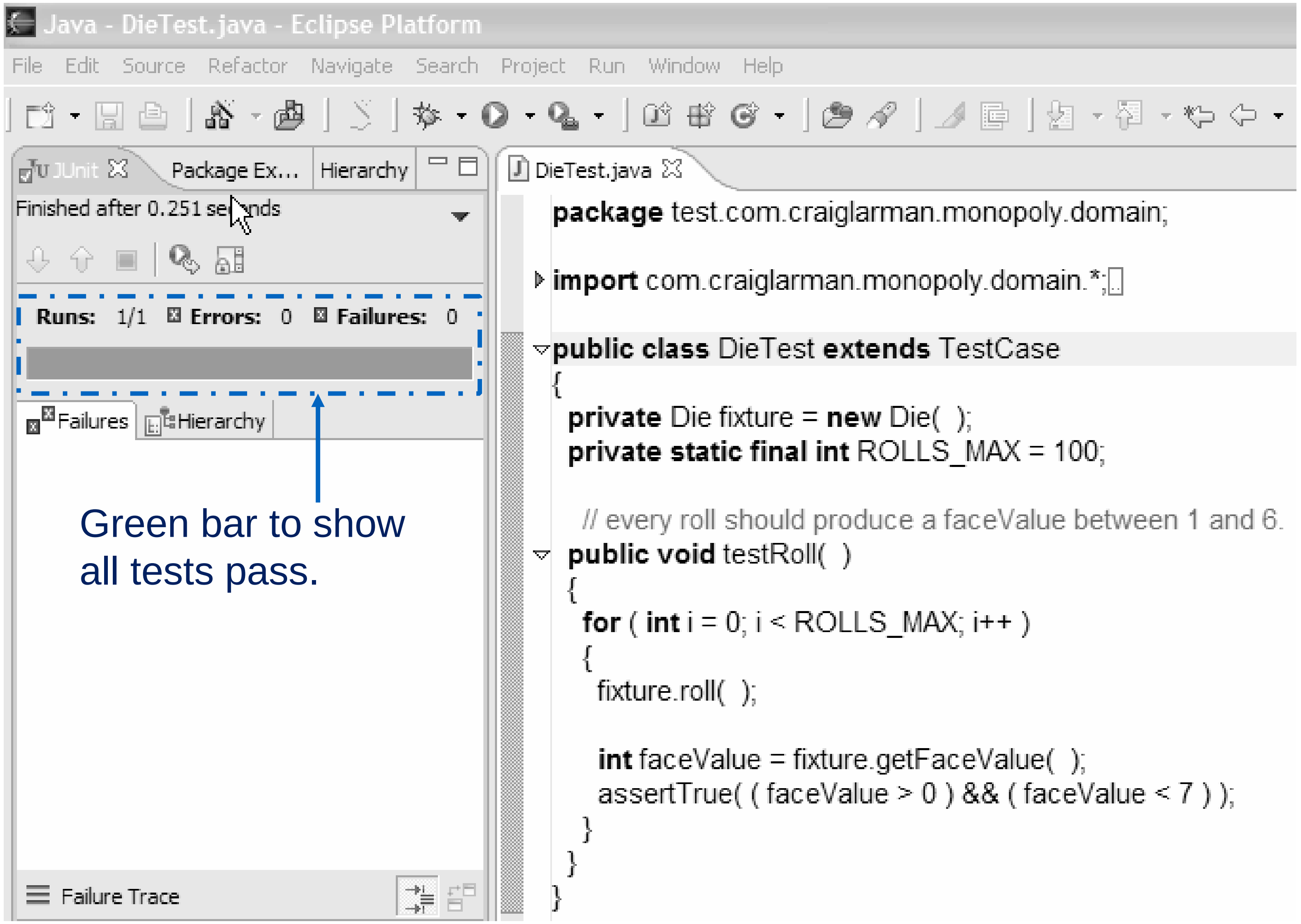
Example: Unit Test the *Sale* Class

- Before writing *Sale*, write the class *SaleTest.*
- Choose a method to implement/test first in *Sale*
  - E.g. *makeLineItem*

SaleTest method *testMakeLineItem* will:

1. Create a *Sale* (test item, aka the *fixture*)
2. Add some line items to it with *makeLineItem*
3. Ask for the total and verify it is as expected using assertions.

IDE Support for Testing


Java - DieTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
JUnit
Package Ex...
Hierarchy
DieTest.java
Failures
Hierarchy
Failure Trace

Finished after 0.251 seconds

Runs: 1/1 Errors: 0 Failures: 0

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

Green bar to show all tests pass.

Refactoring

- Structured, disciplined method for rewriting or restricting existing code *without changing its external behaviour*.
- Small behaviour preserving transformations (*refactors*) can be applied, one at a time.
- Unit tests can be re-executed to show that the refactoring id not cause a regression (failure).
- A series of small test transformations can result in a major restructuring of the code and design (for the better) with no behaviour change.

Bad Smelling Code

Examples:

- 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

Refactorings

There are over 100 named refactorings.

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 *isLeapYear* Method

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

isLeapYear with Explaining Vars

Explaining Variables

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

The *takeTurn* Method

```
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 *takeTurn* Method after Extract Method

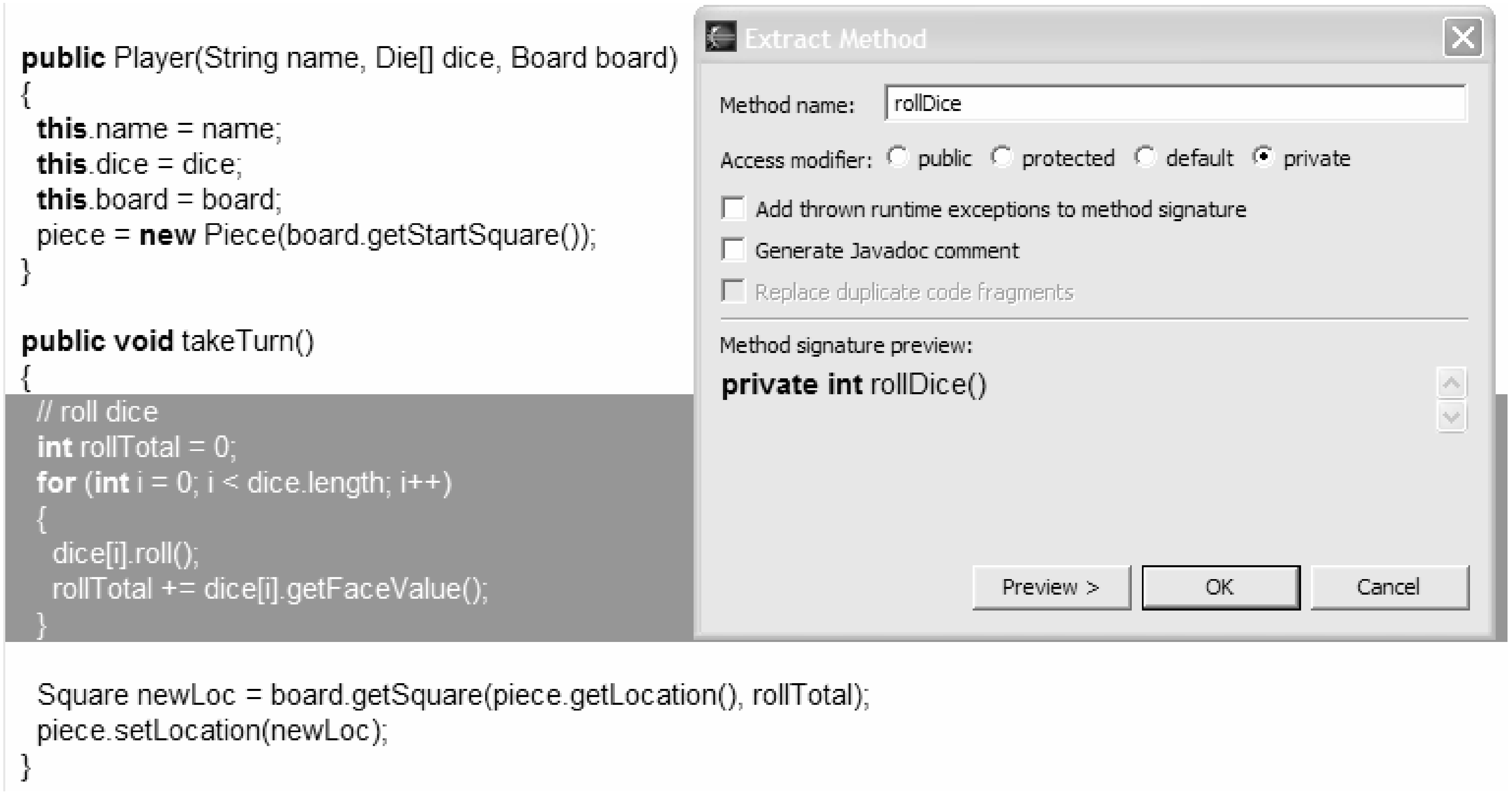
```
public void takeTurn()
{
               // the refactored helper method
       int rollTotal = rollDice();

       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;
}
```

$\leftarrow$ Extracted Method

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);
}
```

Extract Method

Method name: rollDice

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

☐ Add thrown runtime exceptions to method signature

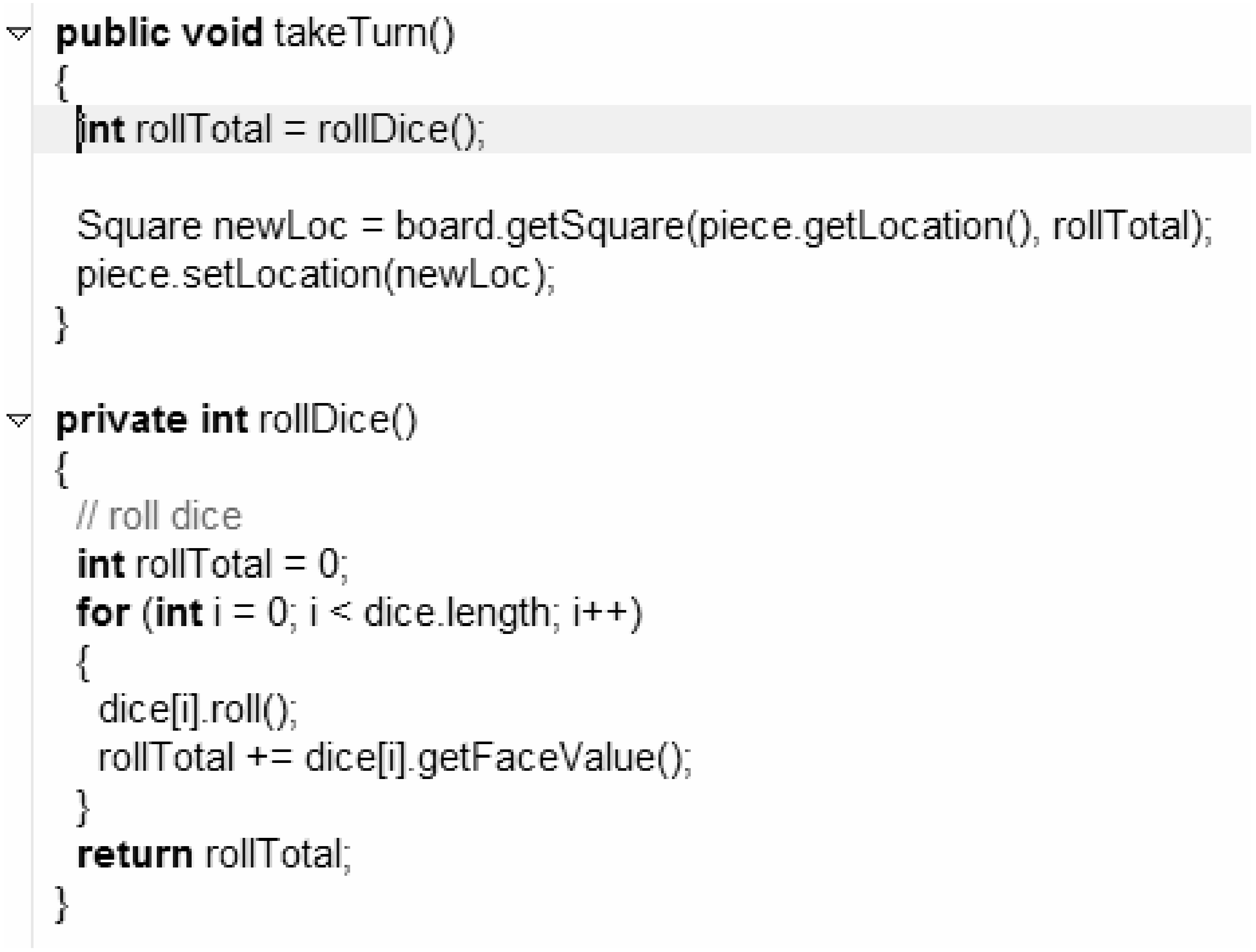
☐ Generate Javadoc comment

☐ 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;
}
```

Conclusion

- Test driven development can play a key role in an Agile process
- A set of successfully passed tests represents a behaviour base-line for the system and its components
- Making design changes can be essential to keeping a maintainable, modifiable and understandable design
- Refactoring, supported by regression testing, is a disciplined approach to achieving design changes