SWEN20003
Object Oriented Software Development

Software Design and Testing

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne

The Road So Far

- Java Foundations
  - A Quick Tour of Java
- Object Oriented Programming Foundations
  - Classes and Objects
  - Arrays and Strings
  - Input and Output
  - *Software Tools and Bagel*
  - Inheritance and Polymorphism
  - Interfaces and Polymorphism
- Advanced Object Oriented Programming and Software Design
  - Modelling Classes and Relationships
  - Generics
  - Collections and Maps
  - Design Patters
  - Exceptions

Lecture Objectives

After this lecture you will be able to:

- **Write better** code
- **Design better** software
- **Test** your software for bugs

Good Coding Practices

Coding Standards

While writing code is largely subjective, there are plenty of **conventions** that most programmers share. Some organizations publish these as **coding standards**:

- Use consistent layout (indentation, white space)
- Avoid long lines (80 characters is "historic")
- Beware of tabs
- Lay out comments and code neatly
- Sensible naming of variables, method and classes
- Avoid copy and pasting/duplicating code
- Use a comment to explain each section

Comment Style

While writing comments is largely subjective, there are plenty of **conventions** that most programmers share:

- Intended primarily for **yourself**, and developers writing code **with** you
- Code should be written to be **self-documenting**; readable without extra documentation
- Comments “tell the story” of the code
- If your code were removed, comments should be sufficient to “piece together” the algorithm
- Comments should be attached to *blocks* of code, which loosely correspond to *steps* in completing your algorithm

Comment Placement

Bad Comment Placement

```
<line of code>
// This is a comment below my code
```

Great Comment Placement

```
// This is a comment above my code
<line of code>
```

Comments appearing **before** code are like a "prologue" for your code; they introduce the *idea* of the code before you actually try to digest it.

Documenting your code - Javadoc

javadoc is a command-line tool, provided with the java development kit, which is able to automatically generate documentation from a special type of comment.

3 types of java comment

① // line comment (every line should start with //)

② /* */

③ /** */

javadoc comments:

- starts with a /** and ends with */
- Can be **compiled to HTML**
- Used to document packages, classes, methods, and attributes (among others)
- Various @ tags (like `@param` and `@return`) for generating specific documentation
- Intended primarily for developers **using** your program
- Should document how to **use** and **interact** with your classes and their methods
- Writing Javadoc is not equivalent to commenting code

Documenting your code - Javadoc example

in Intellj

tools → Generate JavaDoc

```
/** This example demonstrates how to include
* javadoc comments in a program using various tags
* @author  Shanika Karunasekera
* @version 1.0
*/
public class JavaDocExample {
    /** This method is used to add two integers.
    * @param a This is the first paramter to addNum method
    * @param b  This is the second parameter to addNum method
    * @return int This returns sum of numA and numB.
    */
    public int addNum(int a, int b) {
        return a + b;
    }
    /** This is the main method which makes use of addNum method.
    */
    public static void main(String[] args) {
        JavaDocExample jd = new JavaDocExample();
        System.out.println("Sum of numbers is: " + jd.addNum(1,2));
    }
}
```

Project Expectations (Javadoc)

You **must** include Javadoc documentation in your project 2 submission:

- **All** public classes, attributes, and methods
- Yes, this includes getters, setters, and constructors (hint, some things can be auto-generated)
- You **do not** need to use any fancy `@` tags, just provide `@param` and `@return` when appropriate
- No, we're not generating the HTML of your Javadoc

Software Design

Poor Design Symptoms

Think about how the information in the following slides can be applied to your current/expected implementation of Project 2.

Imagine how difficult it would be for you to **change**/**fix**/**update** your solution if you identified a problem, or if the specification changed.

Design Principles

So for we have been learning design concepts in the object oriented context:

- Encapsulation, Information Hiding, Delegation (Association), Inheritance (Generalization), Realization (Interfaces), Polymorphism
- Modelling classes and relationships
- Design Patterns

What you have seen so far are applications of some general *design principles* to the object oriented software development paradigm.

Now let us take a look at some of these general design principles ..

Design Principles

break down to lowest possible unit

Keyword

Modularity: Decomposing the problem to units (modules) that are easy to understand, manage and re-use.

- In the object oriented paradigm classes are the basic modules.
- Classes are then combined through different types of relationships to solve larger scale problems.

Design Principles

dependence within

Keyword

Cohesion: Modules must be designed to solve clear, focused problems. Designs must have **high** cohesion.

- Classes must be defined to have high cohesion.
- The class' methods/attributes are related to, and work towards, a common objective.

Design Principles

between module

Keyword

Coupling: The degree of interaction between modules must be reduced as much as possible. Designs should have **low** coupling.

- Coupling between classes must be reduced as much as possible.
- Deciding when to have an association relationships should be done carefully - avoid unnecessary associations as much as possible because it increases coupling.
- Decide when it is appropriate to pass objects as parameters (dependency relationships), to reduce coupling.
- Interfaces promote low coupling.

Design Principles

Keyword

Open-Closed Principle: Modules should be **open** to extension, but **closed** to modification.

- In practice, this means if we need to *change* or *add* functionality to a class, we should not have to modify the original class.
- Use delegation, inheritance, interfaces and polymorphism.

Design Principles

Keyword modularity

Abstraction: Solving problems by creating *abstract data types* to represent problem components; achieved in OOP through *classes*, which represent data and actions.

Keyword cohesion

Encapsulation: The details of a class should be kept *hidden* or *private*, and the user's ability to access the hidden details is *restricted* or *controlled*. Also known as **data** or **information hiding**.

Design Principles

Keyword open-close principle

Polymorphism: The ability to use an object or method in many different ways; achieved in Java through *ad hoc* (overloading), *subtype* (overriding, substitution), and *parametric* (generics) polymorphism.

Keyword

Delegation: Keeping classes *focused* by passing work to other classes. Computations should be performed in the class with the *greatest amount of relevant information*.

Poor Design Symptoms

Rigidity	Hard to modify the system because changes in one class/method cascade to many others
Fragility	Changing one part of the system causes unrelated parts to break
Immobility	Cannot decompose the system into reusable modules
Viscosity	Writing "hacks" when adding code in order to preserve the design
Complexity	Lots of clever code that isn't necessary right now; premature optimisation is bad
Repetition	Code looks like it was written by Cut and Paste
Opacity	Lots of convoluted logic, design is hard to follow

Software Testing

Bug Fixing

Bug → mainly semantic error

$\boxed{\text{problem in the logic}}$

How do you normally find/fix a bug?

- Print statements

```
System.out.println("Why does my code not reach here?");
```

- Google

```
How to fix my Java code
```

- Forums (Stackoverflow, etc.)

```
Someone please help my code is broken
```

Software Testing

Bugs in software are discovered through **Software Testing**.

Phases of software testing:

bottom-up testing strategy

- **Unit testing**: testing units/components independently before integrating (combining)
- **Integration and system testing**: integrating units to form the system and testing the system as a whole
- **Acceptance testing**: validating if the system support the functionality as per requirements

Software Testing

Java offers a structured method for **unit testing**.

class
method
submethod ..

Keyword

Unit: A small, well-defined component of a software system with one, or a small number, of responsibilities.

Keyword

Unit Test: Verifying the operation of a *unit* by testing a single *use case* (input/output), intending for it to **fail**.

Keyword

Unit Testing: Identifying bugs in software by subjecting every *unit* to a suite of *tests*.

Test Cases

What **test cases** can you think of for the following method?

```java
public boolean makeMove(Player player, Move move) {

    int row = move.row;
    int col = move.col;

    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||
            !board[row][col].equals(EMPTY)) {
        return false;
    }

    board[row][col] = player.getCharacter();

    return true;

}
```

⇒ check valid move

Test Cases

Test the method for:

- Valid input
- Invalid input

Great... But that's not helpful

Test Cases

Test the method for::

- Valid input
  - Does a move with row and column on the board...
    - Change the right position on the board?
    - Does the right character get used?
    - Does the method return true in this case?
- Invalid input
  - Does a move that is not on the board do nothing?
  - Does a move do nothing if the position is full?
  - Does the method return false in these cases?

Another Look

```
public boolean makeMove(Player player, Move move) {

    int row = move.row;
    int col = move.col;

    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||
            !board[row][col].equals(EMPTY)) {
        return false;
    }

    board[row][col] = player.getCharacter();

    return true;
}
```

How could we *better abstract* this code to make testing easier?

Creating Units

What are the fundamental *units* of this method?

```java
public boolean makeMove(Player player, Move move) {

    int row = move.row;
    int col = move.col;

    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||
            !board[row][col].equals(EMPTY)) {
        return false;
    }

    board[row][col] = player.getCharacter();

    return true;
}
```

→ check for validity

→ move

Creating Units

```java
public boolean cellIsEmpty(Move move) {
    return board[move.row][move.col].equals(EMPTY);
}
```

```java
public boolean onBoard(Move move) {
    return move.row >= 0 && move.row < SIZE &&
            move.col >= 0 && move.col < SIZE;
}
```

```java
public boolean isValidMove(Move move) {
    if (onBoard(move) && cellIsEmpty(move)) {
        return true;
    }
    return false;
}
```

```java
public void makeMove(Player player, Move move) {
    board[move.row][move.col] = player.getCharacter();
}
```

Creating Units - Improved code

```java
public boolean makeMove(Player player, Move move) {
    if (!isValidMove(move)) {
        return false;
    }

    makeMove(player, move);

    return true;

}
```

Unit Testing With Java

Much better! What now?

Keyword

临时的

Manual Testing: Testing code manually, in an ad-hoc manner. Generally difficult to reach all edge cases, and not scalable for large projects.

Keyword

Automated Testing: Testing code with automated, purpose built software. Generally faster, more reliable, and less reliant on humans.

JUnit Automated Testing

Keyword

assert: A true or false statement that indicates the success or failure of a test case.

Keyword

TestCase class: A class dedicated to testing a single unit.

Keyword

TestRunner class: A class dedicated to *executing* the tests on a unit.

TestCase Class

```
import static org.junit.Assert.*;
import org.junit.Test;
public class BoardTest {
    @Test
    public void testBoard() {
        Board board = new Board();
        assertEquals(board.cellIsEmpty(0, 0), true);
    }
    @Test
    public void testValidMove() {
        Board board = new Board();
        Move move = new Move(0, 0);
        assertEquals(board.isValidMove(move), true);
    }
    @Test
    public void testMakeMove() {
        Board board = new Board();
        Player player = new HumanPlayer("R");
        Move move = new Move(0, 0);
        board.makeMove(player, move);
        assertEquals(board.getBoard()[move.row][move.col], "r");
    }
}
```

→ check (0,0) is empty

→ check (0,0) is a Valid position to move

TestRunner Class

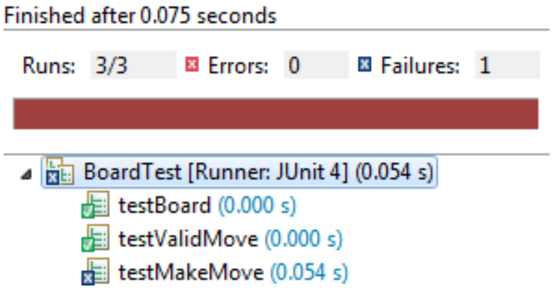
```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(BoardTest.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

TestRunner Class




```
testMakeMove(BoardTest): expected:<[r]> but was:<[R]>
false
```

JUnit Automated Testing

Woops, there was a *bug in my test*

```
@Test
public void testMakeMove() {
    Board board = new Board();
    Player player = new HumanPlayer("R");
    Move move = new Move(0, 0);
    board.makeMove(player, move);
    assertEquals(board.getBoard()[move.row][move.col], "R");
}
```

Automated testing is as useful for testing your *test suite* as it is for testing your *program*.

Assess Yourself

Write a unit test to verify that when a move is made **off the board**, the `isValidMove` method returns false.

There are actually (at least) **four** test cases for this, but here's one:

```java
@Test
public void testValidMove2() {
    Board board = new Board();
    Move move = new Move(-1, 0);
    assertEquals(false, board.isValidMove(move));
}
```

JUnit Advantages

Large teams and open source development **(should) always** use automated testing:

- Easy to set up
- Scalable
- Repeatable
- Not human intensive
- Incredibly powerful
- **Finds bugs**

We don't expect you to use it, but getting used to automated testing makes you more useful in a team.

Here's an example.

Assess Yourself

What units, use cases, and unit tests *could* you write for Project 2B?

Again, we don't expect you to do this.

What you are expected to know for the subject

Documentation
This will be assessed in the project, but not the exam.

Software Design
You will need to be able to define the *keywords* defined in this lecture. You will need to know *definitions* for the exam, but you will not be assessed on software design principles by writing code.

Software Testing
You will need to be able to define the keywords as well as implement a *unit test* for the exam. You will **not** be asked to write a `TestRunner` class, only one or two standalone test cases.

Lecture Objectives

After this lecture you will be able to:

- **Write better** code
- **Design better** software
- **Test** your software for bugs