CS2030 Lecture 2

Testability in Object-Oriented Programming

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2019 / 2020

Lecture Outline

- □ Testing classes using JShell
- □ Writing method tests as method chains
- □ Effects of testing accessors and mutators
- □ Immutability
- □ Bottom-up testing of classes
- □ Factory methods
- □ Introduction to OOP principle of inheritance
  - – Super–sub (Parent–child) classes
  - – is-a relationship
  - – Overriding methods
- □ Cyclic dependency

Testing the **`Point`** class

- How to test a class (say **`Point`**) without using a client?

```
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distance(Point otherpoint) {
        double dispX = this.x - otherpoint.x;
        double dispY = this.y - otherpoint.y;
        return Math.sqrt(dispX * dispX + dispY * dispY);
    }

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }
}
```

JShell as a “Testing Framework”

- □ `JShell` was introduced in Java 9 to provide an interactive shell
  - – uses REPL to provide an immediate feedback loop

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> /open Point.java

jshell> Point p = new Point(1.0, 2.0)
p ==> Point@2b98378d

jshell> p.getX()
$3 ==> 1.0

jshell> /exit
|  Goodbye
```

- □ JShell can be used for unit or integrated (incremental) testing

Writing Tests as Method Chains

- A test can be written as a single method chain, e.g.

```
jshell> new Point(1.0, 2.0).getX()
$2 ==> 1.0

jshell> new Point(1.1, 2.2).getY()
$3 ==> 2.2
```

- Notice that the result is independent of the ordering of the tests

```
jshell> new Point(1.1, 2.2).getY()
$2 ==> 2.2

jshell> new Point(1.0, 2.0).getX()
$3 ==> 1.0
```

- Being able to construct independent tests is a desirable characteristic of software testing

Mutators and its effect on Testing

- When invoking an accessor, such as `getX()`, it is generally assumed that the internal properties of the object would not change, nor have any effect on the state of the program
- Now, consider adding mutators to the `Point` class

```java
public void setX(double x) {
    this.x = x;
}

public void setY(double y) {
    this.y = y;
}
```

- Clearly, `new Point(1.0, 2.0).setX()` would not return a value
- It is desirable that each method returns an object, so as to support method chaining
  - `void` methods should be avoided

Mutators and its effect on Testing

□ Define mutators to return the object

```
public Point setX(double x) {
    this.x = x;
    return this;
}

public Point setY(double y) {
    this.y = y;
    return this;
}
```

□ Method chains can now be constructed

```
jshell> new Point(1.0, 2.0).setX(3.0).getX()
$2 ==> 3.0

jshell> new Point(1.0, 2.0).setX(3.0).getY()
$2 ==> 2.0
```

Mutators and its effect on Testing

- One can set-up a test by assigning the reference of a point object to a `Point` variable and test via that variable

```
jshell> Point p = new Point(1.0, 2.0)
p ==> Point@2b98378d

jshell> p.getX()
$3 ==> 1.0

jshell> p.setX(3.0).getX()
$4 ==> 3.0

jshell> p.setX(3.0)
$5 ==> Point@2b98378d
```

- Notice that throughout the above, `p` maintains the reference to the same object, but the property of the object has changed

Mutators and its effect on Testing

□ Moreover, consider the following `incX` method

```java
public Point incX(double dx) {
    this.x = this.x + dx;
    return this;
}
```

```
jshell> Point p = new Point(1.0, 2.0)
p ==> Point@2b98378d

jshell> p.incX(0.5).getX()
$3 ==> 1.5

jshell> p.incX(0.5).getX()
$4 ==> 2.0
```

□ Clearly, the same test `p.incX(0.5).getX()` returns different values as it depends on some "internal" state of the object

Immutability

- □ Once an object is instantiated, it should not be modified
- □ Ensure by making all instance fields **final**

```
public class Point {
    private final double x;
    private final double y;
```

- □ Notice that this makes the program uncompilable as a statement like `this.x = x` violates immutability
- □ Methods should return other immutable objects

```
public Point setX(double x) {
    return new Point(x, this.y);
}

public Point incX(double dx) {
    return new Point(this.x + dx, this.y);
}
```

Immutability

```
jshell> Point p = new Point(1.0, 2.0)
p ==> Point@2b98378d

jshell> p.setX(3.0).getX()
$3 ==> 3.0

jshell> p.getX()
$4 ==> 1.0

jshell> p.incX(0.5).getX()
$5 ==> 1.5

jshell> p.incX(0.5).getX()
$6 ==> 1.5
```

□ Since `p` references an immutable objecT, `p.setX(..).getX()` (and `p.incX(..).getX()`) will return the same value

Printing a `Point` Object

- Rather than using accessor methods to give details of properties, a `Point` object can simply be output as:

```
jshell> Point p = new Point(1.0, 2.0)
p ==> (1.0, 2.0)
```

- To do this, define an *overriding* `toString` method

```
@Override
public String toString() {
    return "(" + this.x + ", " + this.y + ")";
}
```

- Overrides the same method that is inherited from a parent `Object` class; all classes in Java inherit from the `Object` class
- The annotation `@Override` indicates to the compiler that the method overrides another one

Point Class For Disc Coverage Problem

```java
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distanceTo(Point otherpoint) {
        double dispX = this.x - otherpoint.x;
        double dispY = this.y - otherpoint.y;
        return Math.sqrt(dispX * dispX + dispY * dispY);
    }

    @Override
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}
```

□ Try writing the tests for the `distanceTo` method

Bottom-up Testing

```java
public class Circle {
    private final Point centre;
    private final double radius;

    public Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    public boolean contains(Point point) {
        return centre.distanceTo(point) < radius + 1E-15;
    }

    @Override
    public String toString() {
        return "Circle centered at " + this.centre +
            " with radius " + radius;
    }
}
```

```
jshell> new Circle(new Point(1.0, 2.0), 3.0)
$3 ==> Circle centered at (1.0, 2.0) with radius 3.0

jshell> new Circle(new Point(0.0, 0.0), 1.0).contains(new Point(0.0, 0.0))
$4 ==> true

jshell> new Circle(new Point(0.0, 0.0), 1.0).contains(new Point(1.0, 1.0))
$5 ==> false
```

Testing the `Circle` Class

- What about the following test?

```
jshell> new Circle(new Point(0.0, 0.0), -1.0)
$6 ==> Circle centered at (0.0, 0.0) with radius -1.0
```

- To prevent the creation of invalid objects, **`static`** factory methods can be used to check the validity of the input parameters before generating the object

```
static Circle getCircle(Point centre, double radius) {
    if (radius > 0)
        return new Circle(centre, radius);
    else
        return null;
}
```

Factory Method

□ Factory methods call the constructors to instantiate objects only if the parameters are valid, else a **`null`** value* is returned

□ As such, constructors should not be made accessible to clients, i.e. need to make constructors **`private`**

```
jshell> new Circle(new Point(0.0, 0.0), 1.0)
|  Error:
|  Circle(Point,double) has private access in Circle
|  new Circle(new Point(0.0, 0.0), 1.0)
|  ^----------------------------------^

jshell> Circle.getCircle(new Point(0.0, 0.0), 1.0)
$3 ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> Circle.getCircle(new Point(0.0, 0.0), -1.0)
$4 ==> null
```

** Although returning a **`null`** is still undesirable, let's live with it for now..*

Factory Method

□ For the unit-disc coverage problem, need only define a `getUnitCircle` factory method

```
static Circle getUnitCircle(Point centre) {
    return new Circle(centre, 1.0);
}
```

```
jshell> Circle.getUnitCircle(new Point(0.0, 0.0))
$3 ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> Circle.getUnitCircle(new Point(0.0, 0.0)).contains(new Point(0.0, 0.0))
$4 ==> true

jshell> Circle.getUnitCircle(new Point(0.0, 0.0)).contains(new Point(1.0, 1.0))
$5 ==> false
```

`UnitCircle` as a Sub-Class of `Circle`

- □ Since a unit circle is just a type of circle, the **is-a** relationship indicates the use of another object-oriented principle, namely **inheritance**
  - – **is-a** relationship: `UnitCircle` is a `Circle`
  - – `Circle` is the parent(super) class, while `UnitCircle` is the child(sub) class

```java
public class UnitCircle extends Circle {
    public UnitCircle(Point centre) {
        super(centre, 1.0);
    }
}
```

Inheritance

- □ Notice the sub-class `UnitCircle` invokes the parent's `Circle`'s constructor using **`super`**`(centre, radius)` within it's own constructor
  - – `Circle` constructor must not be made accessible from the sub-class
  - – Modify the accessibility of the constructor to **`protected`**

```
protected Circle(Point centre, double radius) {
    this.centre = centre;
    this.radius = radius;
}
```

- □ If needed, a property of `Circle` (say `radius`)can also be made accessible to the child class by changing the access modifier

```
public class Circle {
    protected final double radius;
```

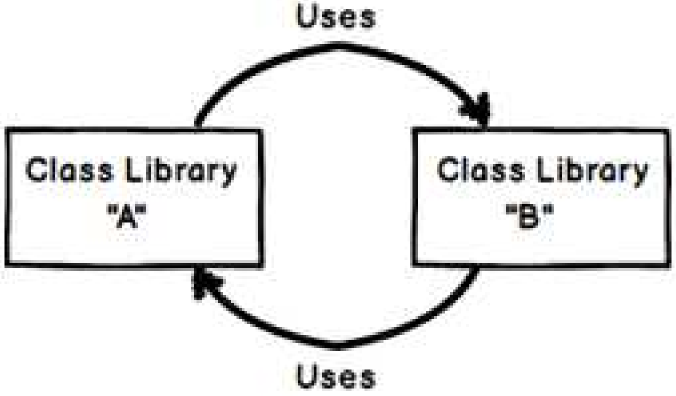
Testing Inheritance

```
jshell> /open Point.java

jshell> /open Circle.java

jshell> /open UnitCircle.java

jshell> new UnitCircle(new Point(1.0, 1.0))
$4 ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(1.0, 1.0))
$5 ==> true

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(2.0, 2.0))
$6 ==> false
```

□ It is worth noting that although instantiations of `Circle` objects are now possible, this issue will be resolved when packages are introduced

Cyclic Dependency

- □ Class dependency in the form of
  - *hard dependencies*: references to other classes in instance fields/variables
  - *soft dependencies*: references to other classes in methods (i.e. parameters, local variables, return type)
- □ Dependencies of classes/components **should not** have cycles
  - Avoid cyclic dependencies, e.g. testing class A requires class B to be tested first, and vice-versa



Cyclic Dependency

□ Using a simplified library system as an example, we would like to model the `Student` and `Book` class

```java
public class Student {
    private final String name;
    private final Book book;

    public Student(String name, Book book) {
        this.name = name;
        this.book = book;
    }

    public String getName() {
        return this.name;
    }

    public String getBookTitle() {
        return this.book.getTitle();
    }
}
```

```java
public class Book {
    private final String title;
    private final Student student;

    public Book(String title, Student student) {
        this.title = title;
        this.student = student;
    }

    public String getTitle() {
        return this.title;
    }

    public String getStudentName() {
        return this.student.getName();
    }
}
```

□ How do we set up a student to borrow a book?

□ How do we perform bottom-up testing?

Cyclic Dependency

- Use an association class to break the cyclic dependency
  - A student borrows a book under a **loan**

```java
public class Student {
    private final String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

public class Book {
    private final String title;

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return this.title;
    }
}
```

```java
public class Loan {
    private final Student student;
    private final Book book;

    public Loan(Student student, Book book) {
        this.student = student;
        this.book = book;
    }

    public String getBookTitle() {
        return this.book.getTitle();
    }

    public String getStudentName() {
        return this.student.getName();
    }
}
```

Lecture Summary

- □ Murphy's Law: *things that can go wrong, will go wrong*
- □ Objective of testing: *things that can go wrong, don't go wrong*
- □ The more flexible the software is, the more ways that things can go wrong, and the more tests are needed
- □ Appreciate that immutability decreases the flexibility of the software, leading to fewer tests
  - – Preventing internal state changes implies that there are no state transitions to test
- □ Appreciate why we need to break cyclic dependencies, so as to facilitate bottom-up testing
- □ Appreciate how to make software easier to test, maintain and more importantly, to reason