CS2030 Lecture 7

Testability of Classes and Methods

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

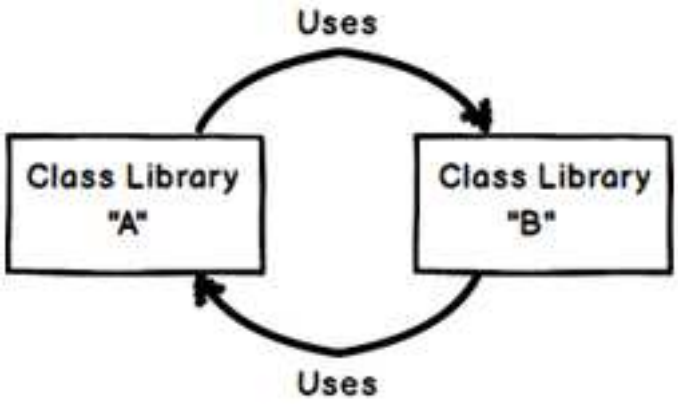
Semester 2 2018 / 2019

Lecture Outline

- Cyclic dependency
- Bottom-up testing of classes
- Method testing
- Immutability
- Dependency Injection via stubbing
- Handling exceptional situations
- Java `Optional` class
- Method chaining

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



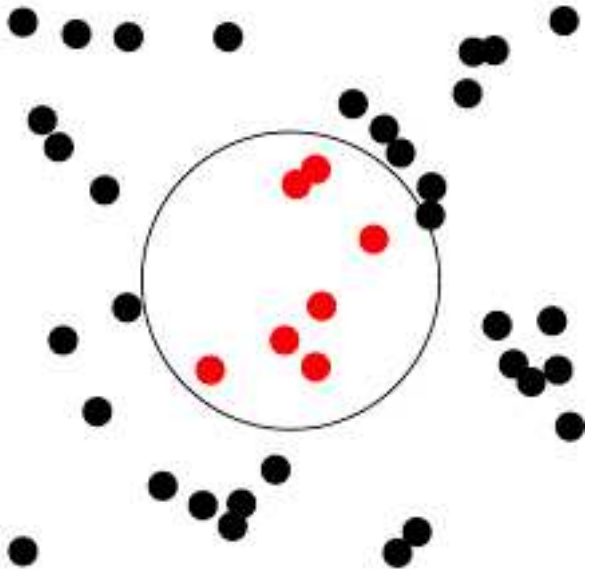
Example: Maximum Coverage Revisited

```
class Circle {
    private Point centre;
    private double radius;

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

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

    @Override
    public String toString() {
       return centre.toString() + ", " + radius;
    }
}
```



A First Attempt

- `Point` class below is functional, but depends on `Circle` class

```
class Point {
   private double x;
   private double y;
   private Circle unitCircle;

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

   Circle createUnitCircle(Point q) {
      Point m = this.midPoint(q);
      double d = Math.sqrt(1 - Math.pow(this.distanceTo(m), 2));
      double theta = this.angleTo(q);

      m.moveTo(theta + Math.PI / 2, d);
      unitCircle = new Circle(m, 1.0);
      return unitCircle;
   }
   ...
```

Class Design for Ease of Testing

- Since `Circle` depends on `Point`, and `Point` depends on `Circle`, which class should we test first?
- Think of *composition*. Which is more logical?
  - Point has a circle? ☹
  - Circle has a point? ☺
- Remove cyclic dependencies
  - remove the hard dependency, i.e. `unitCircle` instance variable in `Point` class
  - remove the soft dependency, i.e. removing the entire `createUnitCircle` method in `Point` class
    - Creating unit circles is a specific application of circles, so it's justifiable to do it in the application class

Bottom-up Testing

- `Point` class can now be tested in isolation

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

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

   double distanceTo(Point p) {
      return Math.sqrt(Math.pow(this.x - p.x, 2) + Math.pow(this.y - p.y, 2));
   }

   Point midPoint(Point p) {
      return new Point((this.x + p.x) / 2, (this.y + p.y) / 2);
   }

   double angleTo(Point p) {
      return Math.atan2(this.y - p.y, this.x - p.x);
   }

   void moveTo(double theta, double d) {
      this.x += d * Math.cos(theta);
      this.y += d * Math.sin(theta);
   }
   ...
```

Method Testing in `Point` Class

- Test the constructor: `new Point(1, 1)`
  - Output instance properties (using `println` or `jshell`) by overriding `toString` method
- Test `distanceTo`/`angleTo` methods:

```
new Point(0,0).distanceTo(new Point(1, 1))
new Point(0,0).angleTo(new Point(1, 1))
```

  - Output is the same each time the above is executed
- Test `midPoint` method:

```
new Point(0,0).midPoint(new Point(1, 1))
```

  - Again, output is the same each time the above is executed
- Deterministic behaviour of method calls is a desirable property

Method Testing

- Test the `moveTo` method:

```
Point p = new Point(0, 0);
p.moveTo(Math.PI / 4, Math.sqrt(2));
```

- Subsequent isolated execution of the instruction `p.moveTo(Math.PI / 3, Math.sqrt(2))` **does not** guarantee that `p` is moved to the same location
- The reason is that `Point` contains **mutable** data!
  - Instance variables `x` and `y` changes (or mutates) every time `moveTo` method is called
  - The **`void`** return type of `moveTo` is indicative that some state changes could have taken place during method invocation

Immutability

- All instance variables once initialized cannot be modified
- Ensure by making all instance fields **final**

```
private final double x;
private final double y;
```

- Replace all **void** methods so that they return other immutable objects

```
Point moveTo(double theta, double d) {
    return new Point(this.x + d * Math.cos(theta),
            this.y + d * Math.sin(theta));
}
```

- Methods that take immutable object parameters *could also* be made **final** (*seems overkill though...*), e.g.

```
double distanceTo(final Point p)
```

Immutable Point Class

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

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

    double distanceTo(final Point p) {
        return Math.sqrt(Math.pow(this.x - p.x, 2) + Math.pow(this.y - p.y, 2));
    }

    final Point midPoint(final Point p) {
        return new Point((this.x + p.x) / 2, (this.y + p.y) / 2);
    }

    double angleTo(final Point p) {
        return Math.atan2(this.y - p.y, this.x - p.x);
    }

    Point moveTo(final double theta, final double d) {
        return new Point(x + d * Math.cos(theta), y + d * Math.sin(theta));
    }

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

Immutable Circle Class

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

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

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

    @Override
    public String toString() {
        return centre.toString() + ", " + radius;
    }
}
```

- Having tested the `Point` class, proceed to test "upwards"

```
new Circle(new Point(0, 0), 1.0)

new Circle(new Point(0, 0), 1.0).contains(new Point(0.5, 0.5))
```

Stubbing

- How to isolate the functionality of `Circle` from that of `Point`?
- *Dependency injection* via inheritance: `PointStub` is-a `Point`

```
class PointStub extends Point {

    PointStub(double x, double y) {
        super(x, y);
    }
    ...
```

- Example: testing `contains` method:

```
new Circle(new PointStub(0, 0), 1.0)
      .contains(new PointStub(0.5, 0.5))
```

- The user/client of `Point` (i.e. `Circle`) does not need to be modified, and `PointStub` extends the functionality of `Point`
  - Open-closed principle

Stubbing

- Another example, creating a `Circle` with two `Points`
- Test `createUnitCircle` using

```
Main.createUnitCircle(new Point(1, 0), new Point(0, 1))
```

```
static Circle createUnitCircle(final Point p, final Point q) {
    Point m = p.midPoint(q);
    double d = Math.sqrt(1 - Math.pow(p.distanceTo(m), 2));
    double theta = p.angleTo(q);

    m = m.moveTo(theta + Math.PI / 2, d);
    Circle unitCircle = new Circle(m, 1.0);
    return unitCircle;
}
```

- Define `PointStub` that caters to individual test cases:

```
Main.createUnitCircle(new PointStub(1, 0, 123),
        new PointStub(0, 1, 123));
```

Creating a Circle with Two Points

```
class PointStub extends Point {

    final int testID;

    PointStub(final double x, final double y) {
        super(x, y);
        this.testID = 0;
    }

    PointStub(final double x, final double y, final int testID) {
        super(x, y);
        this.testID = testID;
    }

    @Override
    Point midPoint(final Point p) {
        switch (testID) {
            case 123:
                return new PointStub(0.5, 0.5);
            default:
                return new PointStub(0, 0);
        }
    }
```

Creating a Circle with Two Points

```
    @Override
    double distanceTo(final Point p) {
        switch (testID) {
            case 123:
                return 1.0 / Math.sqrt(2);
            default:
                return 0.0;
        }
    }

    @Override
    double angleTo(final Point p) {
        switch (testID) {
            case 123:
                return -Math.PI / 4;
            default:
                return 0.0;
        }

    }
}
```

Handling Exceptional Situations

- E.g. handling creation of invalid circles:

```
new Circle(new Point(0, 0), -1.0)
```

- Possible to return **null** from the constructor?
- Define a **static** factory method; make constructor **private**

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

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

   static Circle getCircle(final Point centre, final double radius
      if (radius < 0.0) {
         return null;
      } else {
         return new Circle(centre, radius);
      }
   }
```

Handling Exceptional Situations

- How about points that are too far apart?

```
Main.createUnitCircle(new Point(2, 0), new Point(0, 2))
```

- Using **null**

```
static Circle createUnitCircle(Point p, Point q) {
   Circle unitCircle;

   if (p.distanceTo(q) < 2 + 1E-15 && p.distanceTo(q) > 0) {
      Point m = p.midPoint(q);
      double d = Math.sqrt(1 - Math.pow(p.distanceTo(m), 2));
      double theta = p.angleTo(q);
      m = m.moveTo(theta + Math.PI / 2, d);
      unitCircle = new Circle(m, 1.0);
   } else {
      unitCircle = null;
   }
   return unitCircle;
}
```

Handling Exceptional Situations

- Using `Exception`

```
class Main {
   static Circle createUnitCircle(Point p, Point q) {
      Circle unitCircle;

      if (p.distanceTo(q) < 2 + 1E-15 && p.distanceTo(q) > 0) {
         Point m = p.midPoint(q);
         double d = Math.sqrt(1 - Math.pow(p.distanceTo(m), 2));
         double theta = p.angleTo(q);
         m = m.moveTo(theta + Math.PI / 2, d);
         unitCircle = new Circle(m, 1.0);
      } else {
         throw new IllegalArgumentException("Not a circle");
      }
      return unitCircle;
   }
```

```
jshell> Main.createUnitCircle(new Point(2, 0), new Point(0, 2))
|  Exception java.lang.IllegalArgumentException: Not a circle
|        at Main.createUnitCircle (#3:34)
|        at (#7:1)
```

Handling Exceptional Situations

- What about the following test?

```
Main.createUnitCircle(new Point(2, 0), new Point(0, 2)).toString()
```

- Chaining a `toString` method (or indeed any other method) will result in an exception thrown
- Can `Main.createUnitCircle(new Point(2, 0), new Point(0, 2))` still return a valid object to which further methods can be chained, irrespective of whether a `Circle` is present?
  - The idea is to wrap the object (or absence of the object) around another object! *duh...*
  - Such a "wrapper" object can have a connotation of **maybe**, i.e. the object is maybe a circle, maybe not
- Java's `Optional<T>` has connotations of "maybe"

`Optional` Class

```
import java.util.Optional;

class Circle {
    ...
    static Optional<Circle> getCircle(final Point centre, final double radius) {
        if (radius < 0.0) {
            return Optional.empty();
        } else {
            return Optional.of(new Circle(centre, radius));
        }
    }


class Main {
    ...
    static Optional<Circle> createUnitCircle(final Point p, final Point q) {
        Optional<Circle> unitCircle;

        if (p.distanceTo(q) < 2 + 1E-15 && p.distanceTo(q) > 0) {
            Point m = p.midPoint(q);
            double d = Math.sqrt(1 - Math.pow(p.distanceTo(m), 2));
            double theta = p.angleTo(q);
            m = m.moveTo(theta + Math.PI / 2, d);
            unitCircle = Circle.getCircle(m, 1.0);
        } else {
            unitCircle = Optional.empty();
        }
        return unitCircle;
    }
```

`Optional` Class

- To check the presence of an object in an `Optional`

```
Optional<Circle> c = createUnitCircle(p, q);
if (c.isPresent()) {
   int numOfPoints = findCoverage(c.get(), points);
   if (numOfPoints > maxDiscCoverage) {
      maxDiscCoverage = numOfPoints;
   }
}
```

- To chain methods to an `Optional`

```
jshell> Main.createUnitCircle(new Point(1, 0), new Point(0, 1)).toString()
$11 ==> "Optional[(1.000, 1.000), 1.0]"

jshell> Main.createUnitCircle(new Point(2, 0), new Point(0, 2)).toString()
$12 ==> "Optional.empty"
```

- E.g. returning a default circle if empty

```
jshell> Main.createUnitCircle(new Point(2, 0), new Point(0, 2))
        .orElse(new Circle(new Point(0, 0), 0))
$13 ==> (0.000, 0.000), 0.0
```

Method Chaining

- Invoke multiple method calls in OOP where each method returns an object, allowing the calls to be chained together in a single statement
- Avoid variables for storing intermediate results between method calls
- Methods are called in sequence
  - Useful for testing
  - Makes tests easier to read
- How to ensure immutability of arrays and lists?
- First foray into declarative programming
  - Java streams
  - Functional(-like) programming

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
- Immutability decreases the flexibility of the software, leading to fewer tests
  - Preventing internal state changes implies that there are no state transitions to test
- Passing immutable parameters ensures that methods cannot change these parameters
- Makes the software easier to test, maintain and reason