2014

THAT CONFERENCE

SUMMER CAMP FOR GEEKS ™

@JENNAPEDERSON
WWW.612SOFTWAREFOUNDRY.COM

LEAVE NO TRACE: TEST DRIVEN DEVELOPMENT USING THE SOLID PRINCIPLES

WHO AM I?

- Independent, full-stack web developer
    - Code geek + process geek - agile, TDD, productivity, etc.
- Co-Ambassador for the Twin Cities Geekettes
    - www.geekettes.io
- @jennapederson
- www.612softwarefoundry.com

AGENDA

- What makes a design good?
- Test Driven Development
- SOLID Principles
- Examples

WHAT MAKES A DESIGN GOOD?

IT IS HARD TO CHANGE BECAUSE EVERY CHANGE AFFECTS TOO MANY OTHER PARTS OF THE SYSTEM.

RIGIDITY

WHEN YOU MAKE A CHANGE, UNEXPECTED PARTS OF THE SYSTEM BREAK.

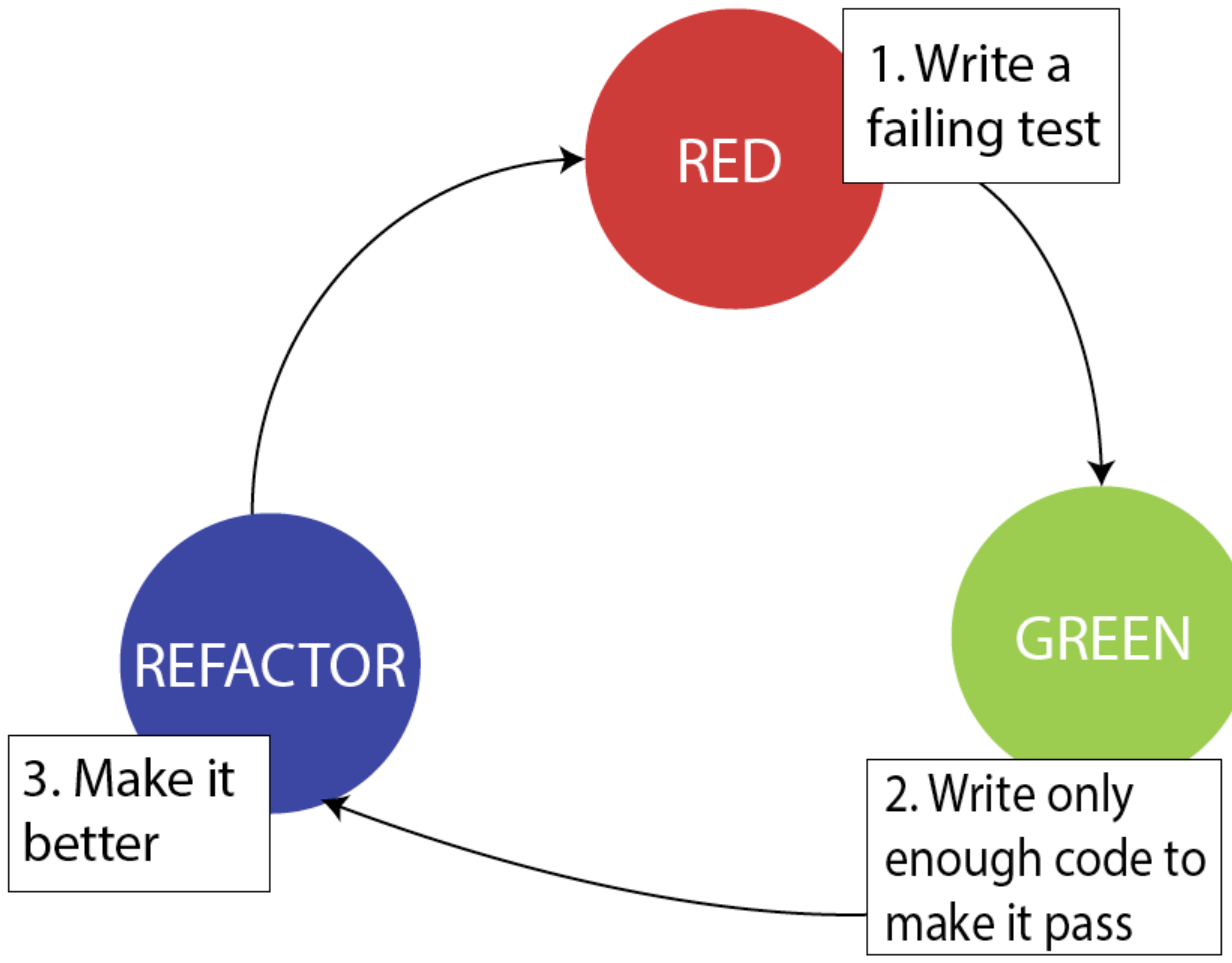
FRAGILITY

IT IS HARD TO REUSE IN ANOTHER APPLICATION BECAUSE IT CANNOT BE DISENTANGLED FROM THE CURRENT APPLICATION.

IMMOBILITY

MAKE IT WORK, MAKE IT BETTER

WHAT IS TDD?


RED
1. Write a failing test
GREEN
2. Write only enough code to make it pass
REFACTOR
3. Make it better

SOLID

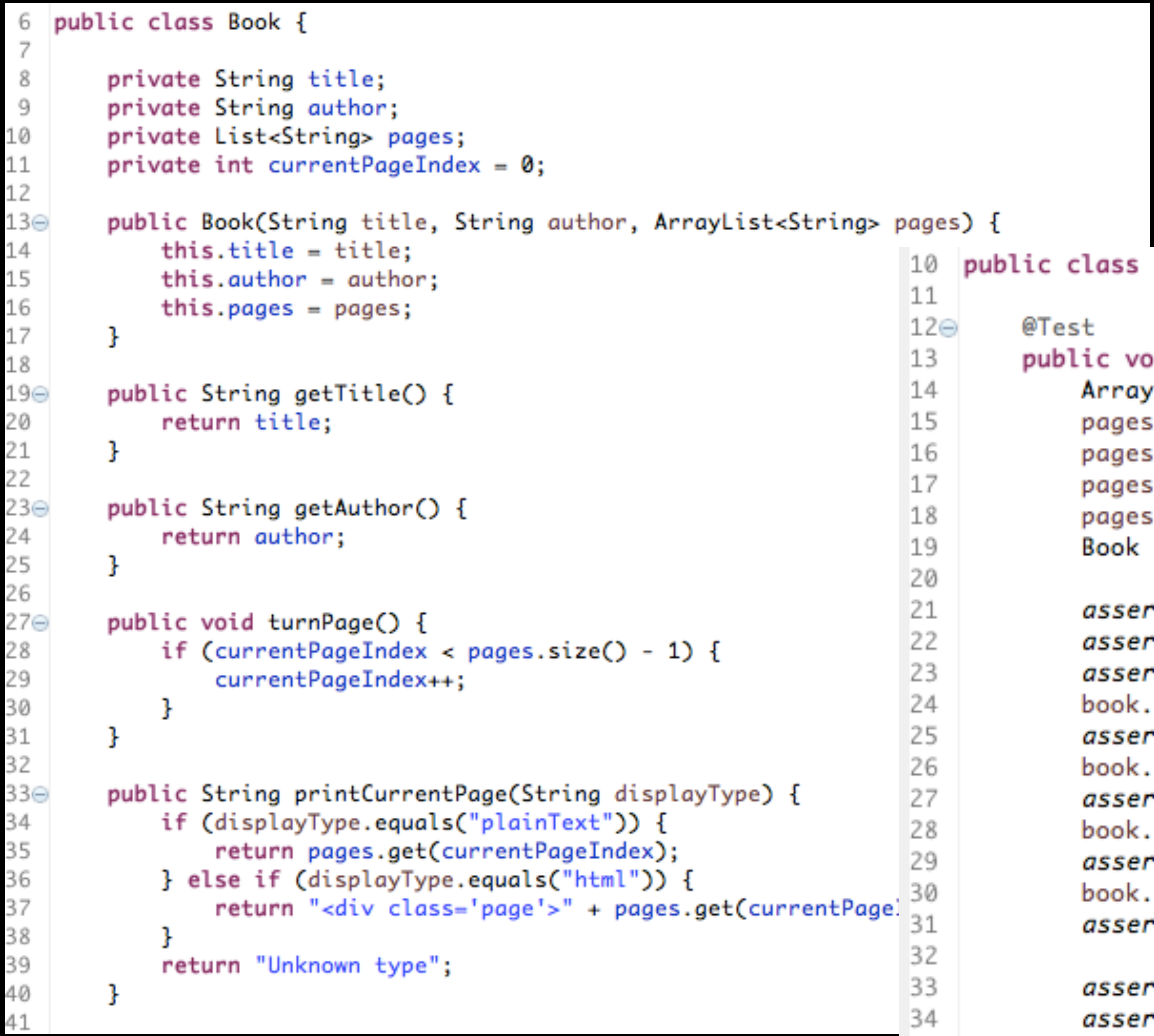
Software Development is not a Jenga game

A CLASS SHOULD HAVE ONE AND ONLY ONE REASON TO CHANGE.

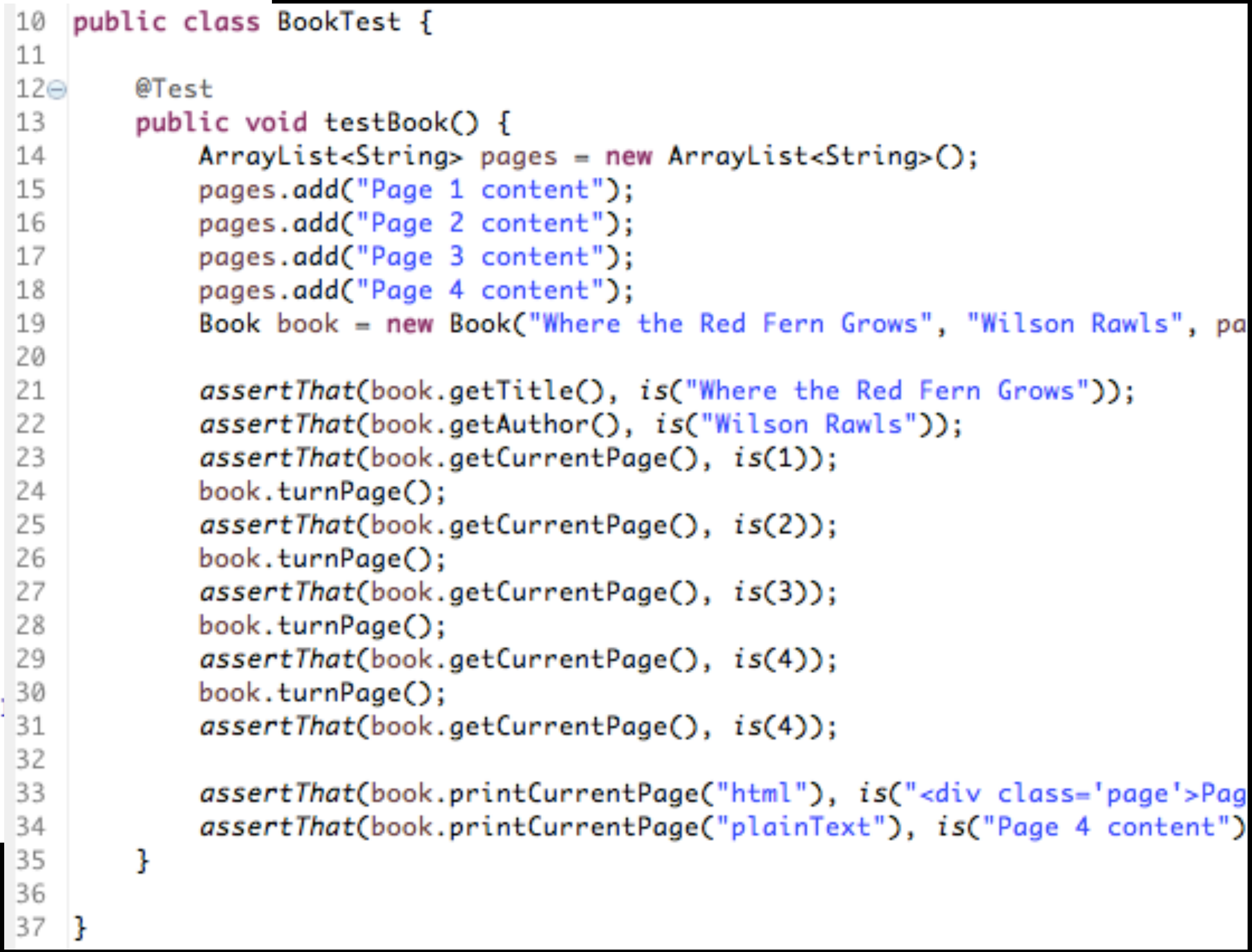
SINGLE RESPONSIBILITY PRINCIPLE

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

BEFORE SRP

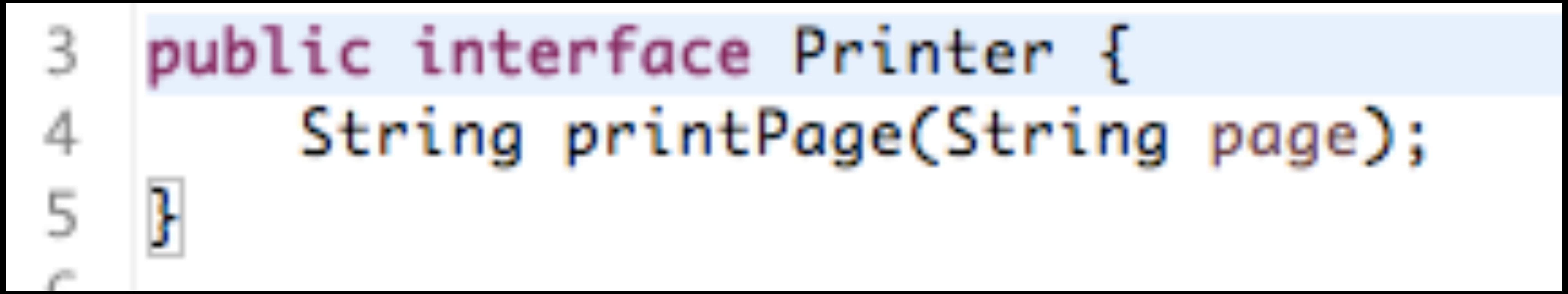


```java
public class Book {

    private String title;
    private String author;
    private List<String> pages;
    private int currentPageIndex = 0;

    public Book(String title, String author, ArrayList<String> pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public void turnPage() {
        if (currentPageIndex < pages.size() - 1) {
            currentPageIndex++;
        }
    }

    public String printCurrentPage(String displayType) {
        if (displayType.equals("plainText")) {
            return pages.get(currentPageIndex);
        } else if (displayType.equals("html")) {
            return "<div class='page'>" + pages.get(currentPage
        }
        return "Unknown type";
    }

```

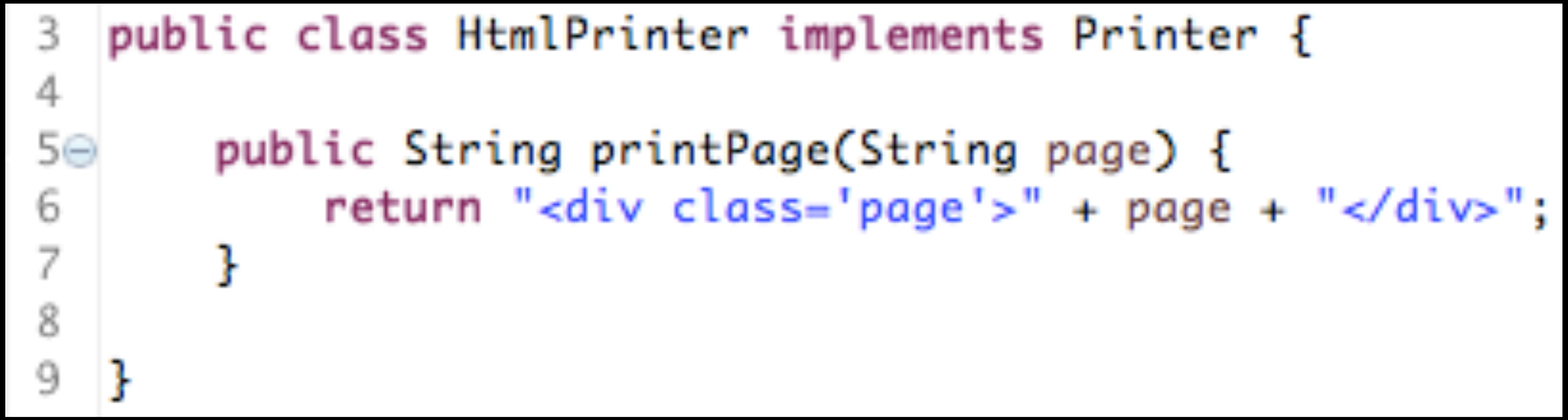


```java
public class BookTest {

    @Test
    public void testBook() {
        ArrayList<String> pages = new ArrayList<String>();
        pages.add("Page 1 content");
        pages.add("Page 2 content");
        pages.add("Page 3 content");
        pages.add("Page 4 content");
        Book book = new Book("Where the Red Fern Grows", "Wilson Rawls", pa

        assertThat(book.getTitle(), is("Where the Red Fern Grows"));
        assertThat(book.getAuthor(), is("Wilson Rawls"));
        assertThat(book.getCurrentPage(), is(1));
        book.turnPage();
        assertThat(book.getCurrentPage(), is(2));
        book.turnPage();
        assertThat(book.getCurrentPage(), is(3));
        book.turnPage();
        assertThat(book.getCurrentPage(), is(4));
        book.turnPage();
        assertThat(book.getCurrentPage(), is(4));

        assertThat(book.printCurrentPage("html"), is("<div class='page'>Pag
        assertThat(book.printCurrentPage("plainText"), is("Page 4 content")
    }

}
```

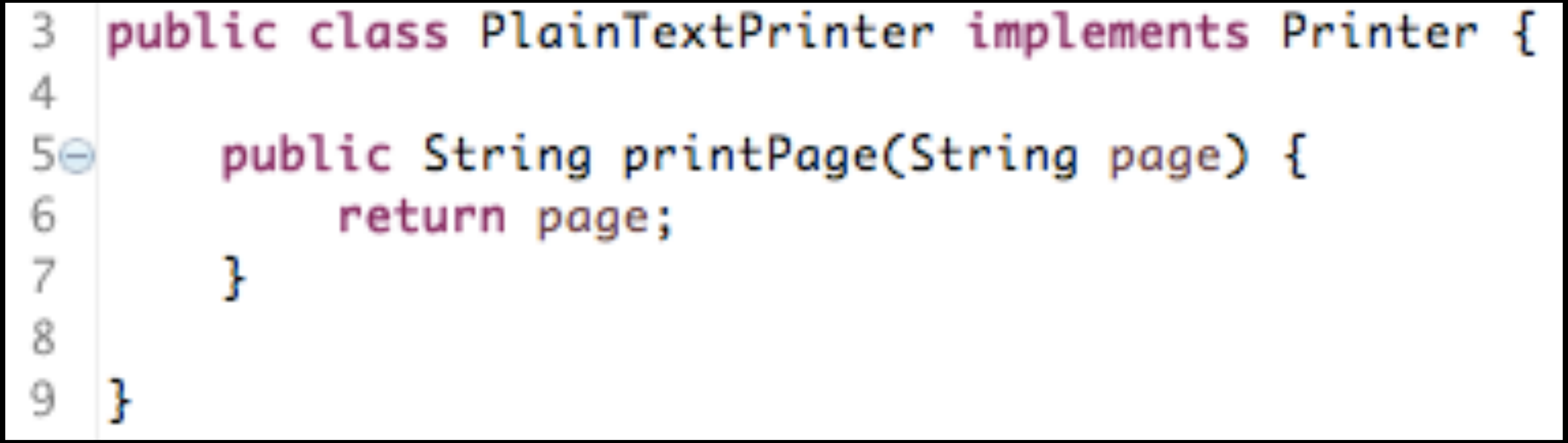
AFTER SRP



```java
public interface Printer {
    String printPage(String page);
}
```

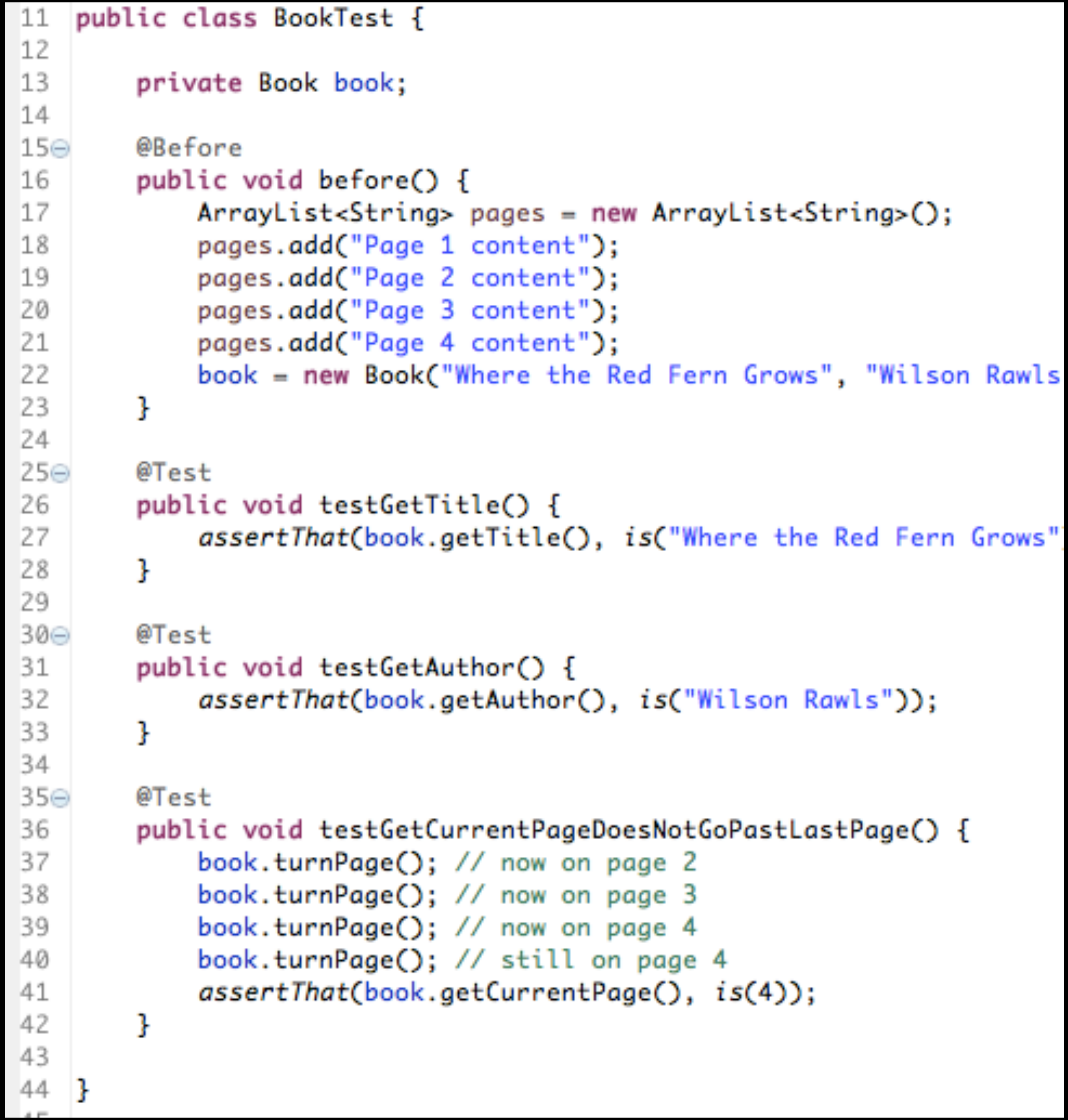


```java
public class HtmlPrinter implements Printer {

    public String printPage(String page) {
        return "<div class='page'>" + page + "</div>";
    }

}
```

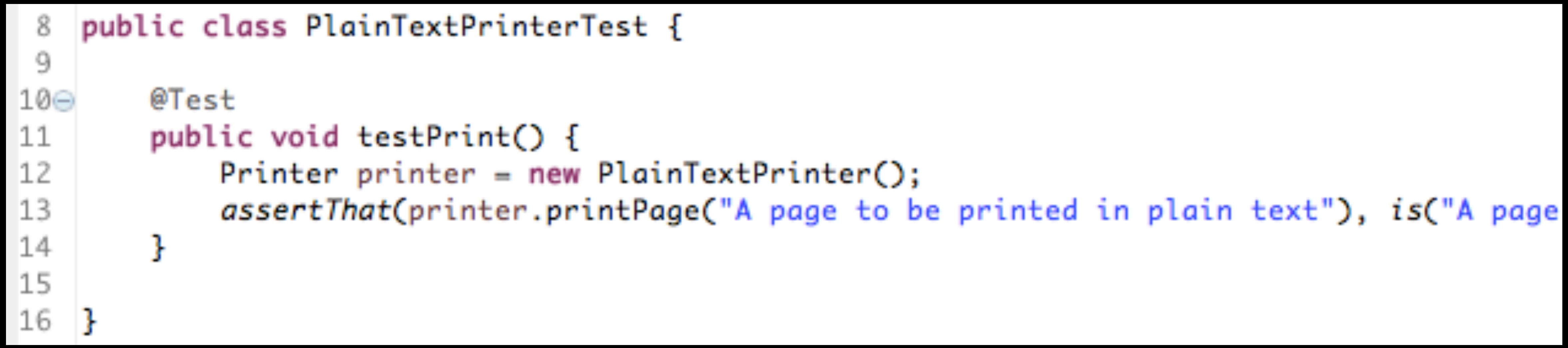


```java
public class PlainTextPrinter implements Printer {

    public String printPage(String page) {
        return page;
    }

}
```

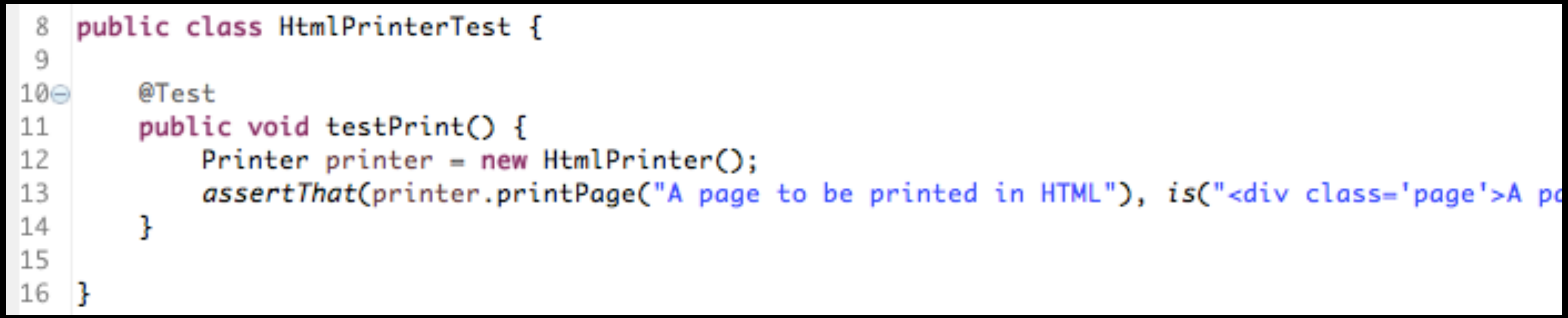
AFTER SRP



```java
public class BookTest {

    private Book book;

    @Before
    public void before() {
        ArrayList<String> pages = new ArrayList<String>();
        pages.add("Page 1 content");
        pages.add("Page 2 content");
        pages.add("Page 3 content");
        pages.add("Page 4 content");
        book = new Book("Where the Red Fern Grows", "Wilson Rawls
    }

    @Test
    public void testGetTitle() {
        assertThat(book.getTitle(), is("Where the Red Fern Grows"
    }

    @Test
    public void testGetAuthor() {
        assertThat(book.getAuthor(), is("Wilson Rawls"));
    }

    @Test
    public void testGetCurrentPageDoesNotGoPastLastPage() {
        book.turnPage(); // now on page 2
        book.turnPage(); // now on page 3
        book.turnPage(); // now on page 4
        book.turnPage(); // still on page 4
        assertThat(book.getCurrentPage(), is(4));
    }

}
```



```java
public class PlainTextPrinterTest {

    @Test
    public void testPrint() {
        Printer printer = new PlainTextPrinter();
        assertThat(printer.printPage("A page to be printed in plain text"), is("A page
    }

}
```



```java
public class HtmlPrinterTest {

    @Test
    public void testPrint() {
        Printer printer = new HtmlPrinter();
        assertThat(printer.printPage("A page to be printed in HTML"), is("<div class='page'>A pa
    }

}
```

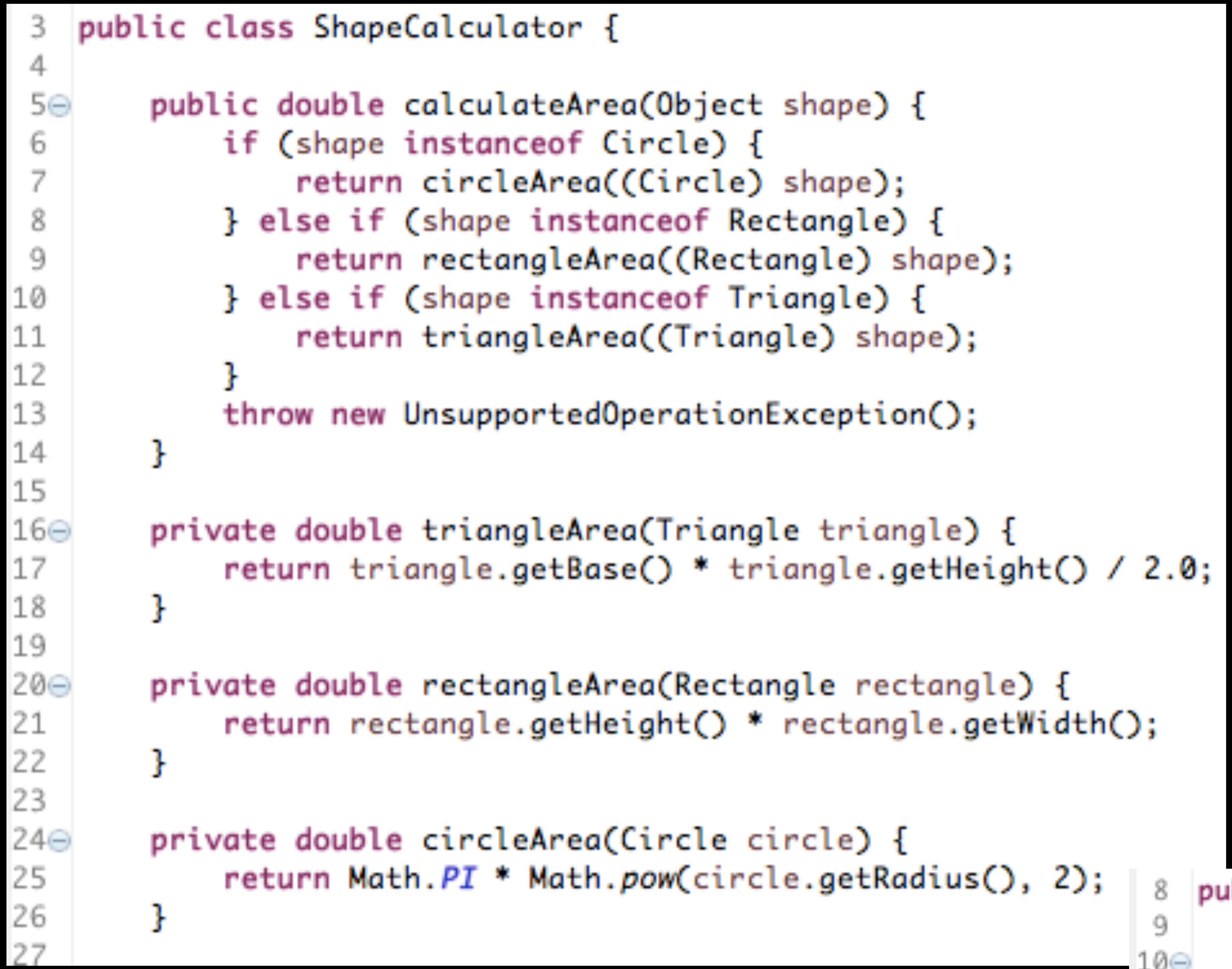
APPLYING SRP TO OUR TESTING

- Tests are more granular
- Easier to understand and maintain
- Fewer changes to tests
- Failing tests point to only one responsibility instead of many

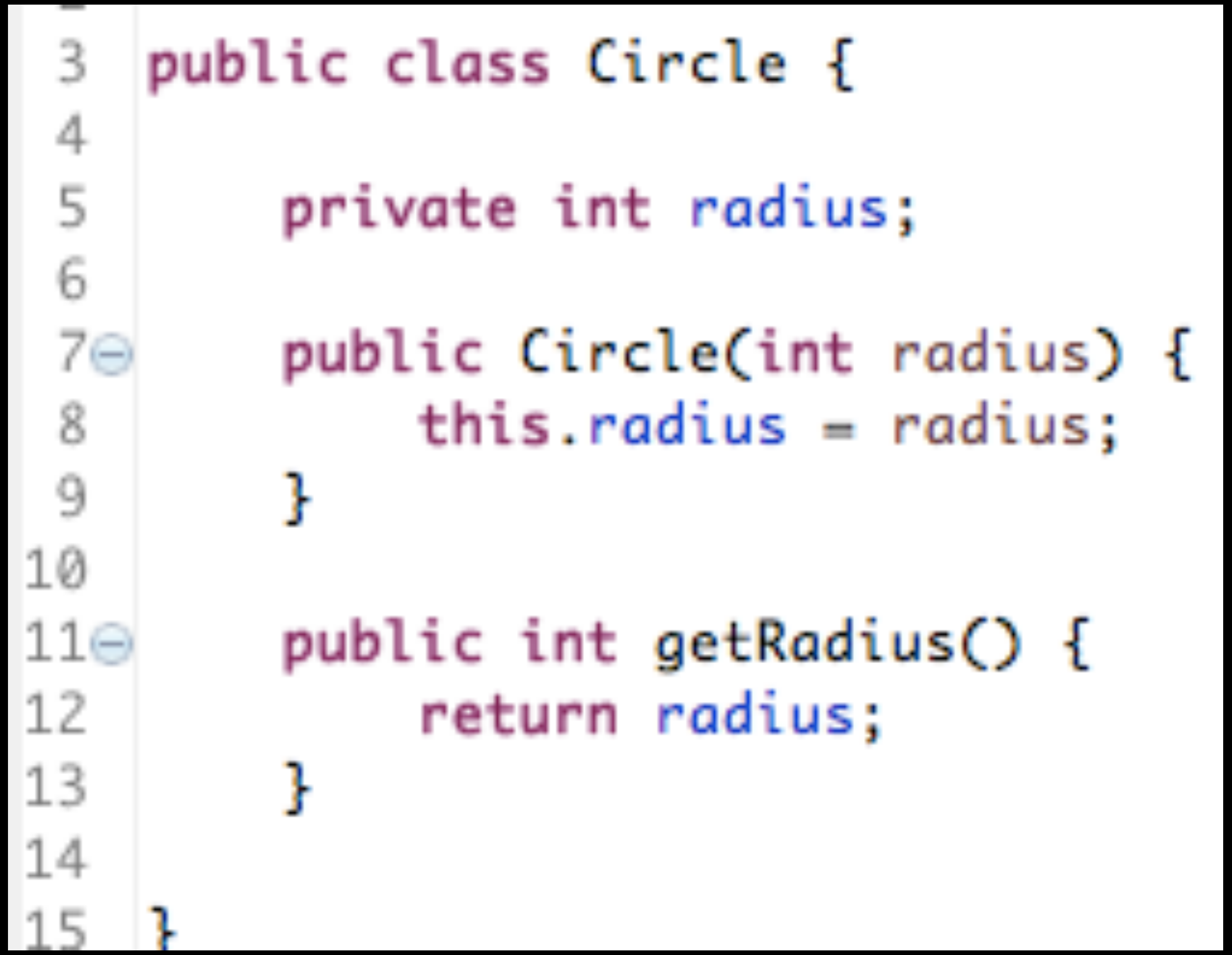
YOU SHOULD BE ABLE TO EXTEND A CLASS' BEHAVIOR, WITHOUT MODIFYING IT.

THE OPEN CLOSED PRINCIPLE

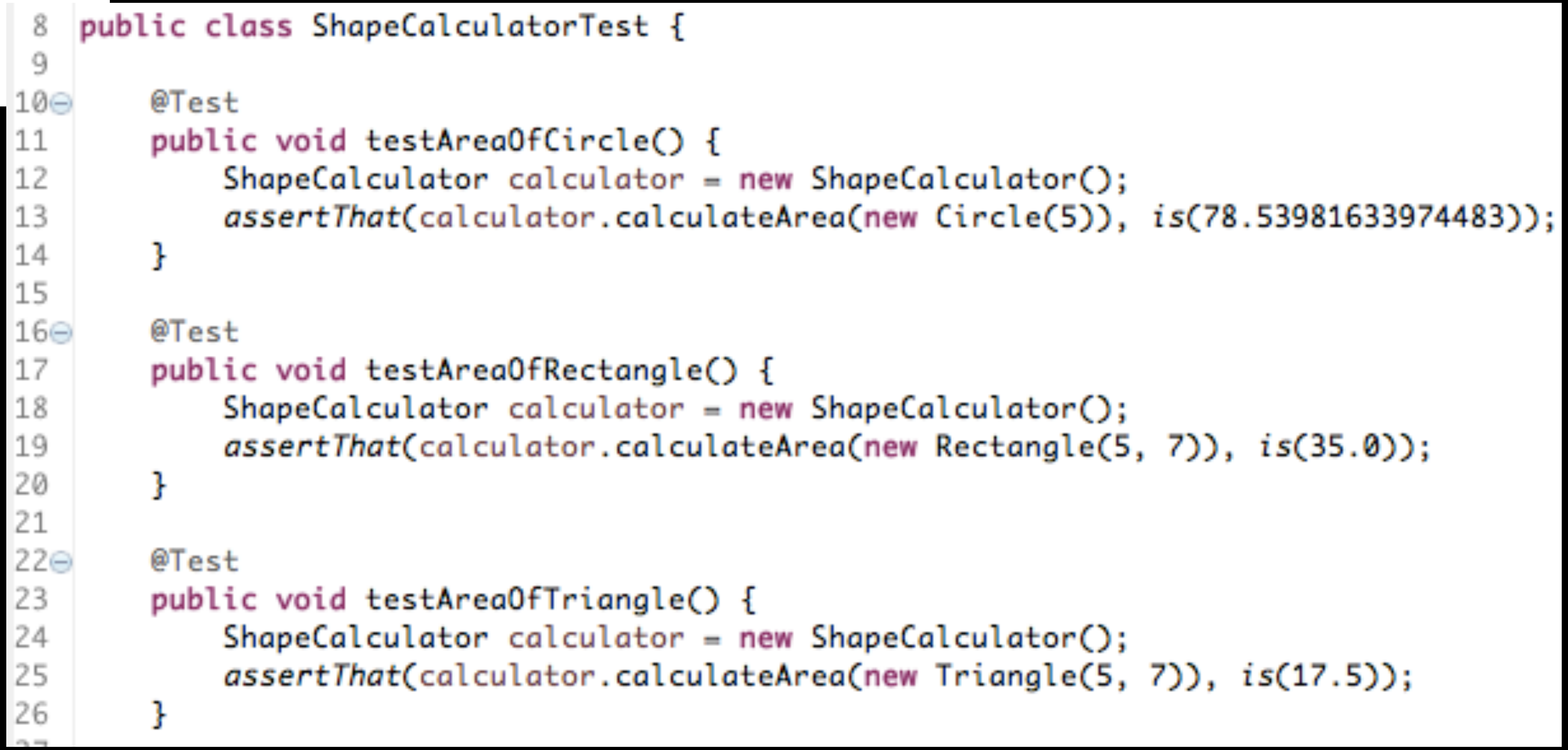
BEFORE OCP



```java
public class ShapeCalculator {

    public double calculateArea(Object shape) {
        if (shape instanceof Circle) {
            return circleArea((Circle) shape);
        } else if (shape instanceof Rectangle) {
            return rectangleArea((Rectangle) shape);
        } else if (shape instanceof Triangle) {
            return triangleArea((Triangle) shape);
        }
        throw new UnsupportedOperationException();
    }

    private double triangleArea(Triangle triangle) {
        return triangle.getBase() * triangle.getHeight() / 2.0;
    }

    private double rectangleArea(Rectangle rectangle) {
        return rectangle.getHeight() * rectangle.getWidth();
    }

    private double circleArea(Circle circle) {
        return Math.PI * Math.pow(circle.getRadius(), 2);
    }

```

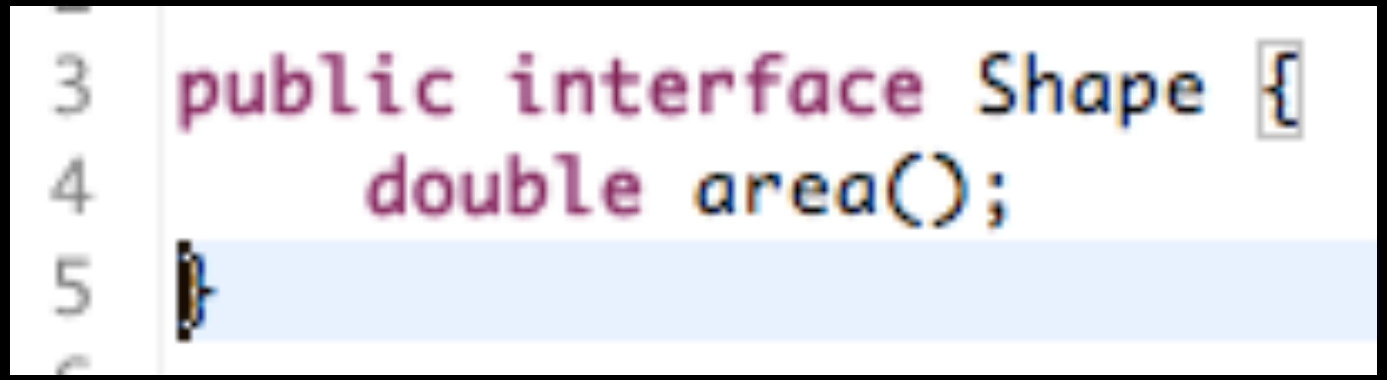


```java
public class Circle {

    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

}
```

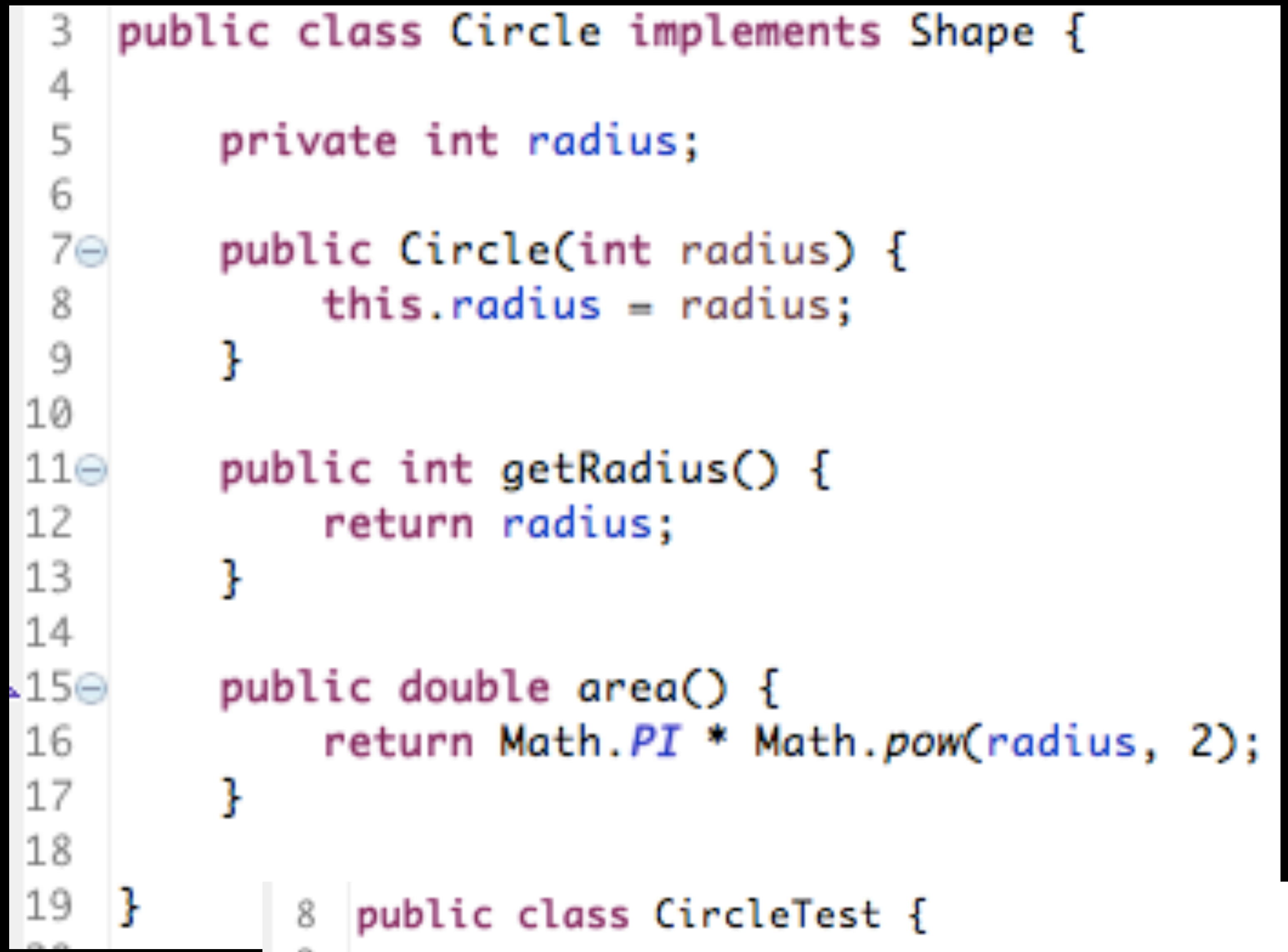


```java
public class ShapeCalculatorTest {

    @Test
    public void testAreaOfCircle() {
        ShapeCalculator calculator = new ShapeCalculator();
        assertThat(calculator.calculateArea(new Circle(5)), is(78.53981633974483));
    }

    @Test
    public void testAreaOfRectangle() {
        ShapeCalculator calculator = new ShapeCalculator();
        assertThat(calculator.calculateArea(new Rectangle(5, 7)), is(35.0));
    }

    @Test
    public void testAreaOfTriangle() {
        ShapeCalculator calculator = new ShapeCalculator();
        assertThat(calculator.calculateArea(new Triangle(5, 7)), is(17.5));
    }
```

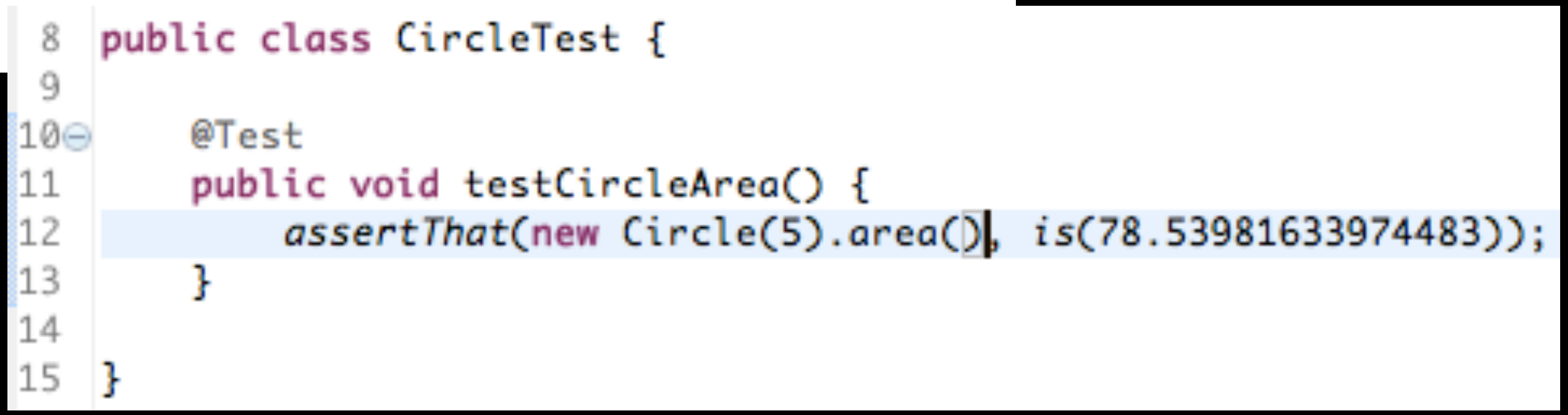
AFTER OCP



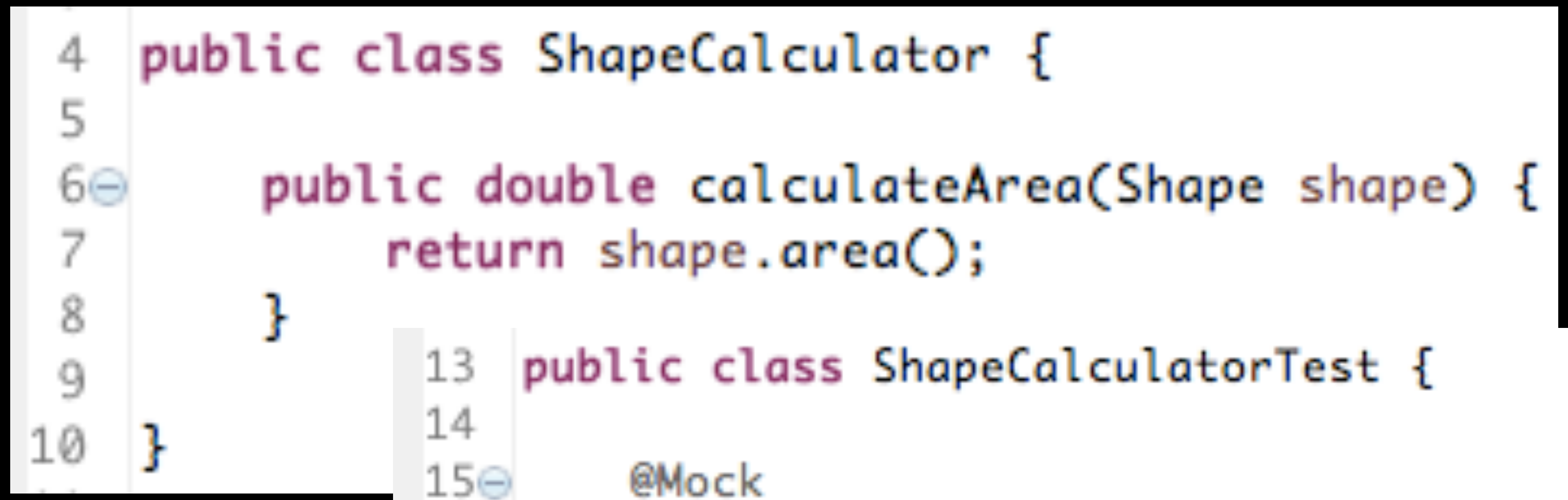
```java
public interface Shape {
    double area();
}
```



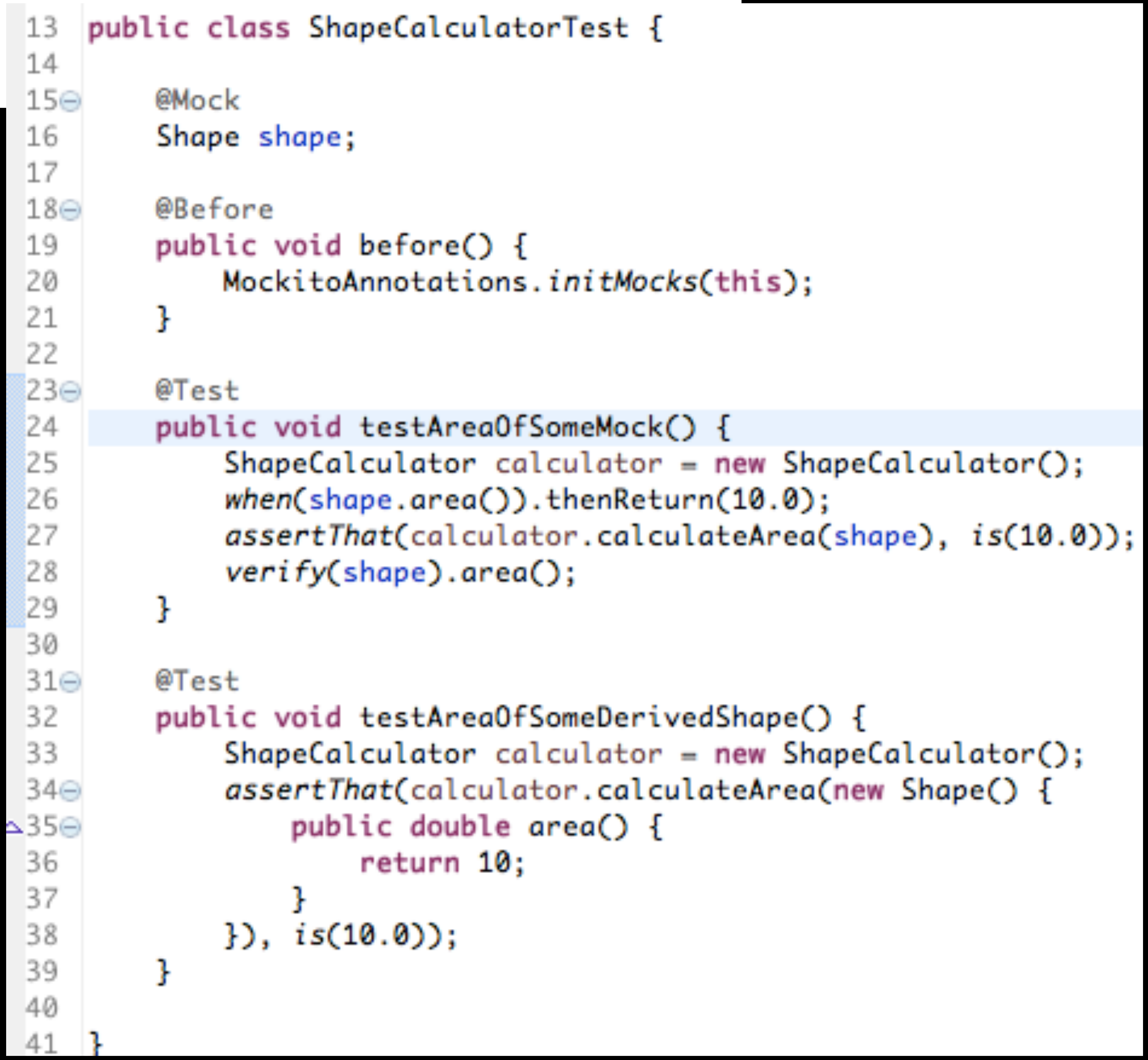
```java
public class Circle implements Shape {

    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    public double area() {
        return Math.PI * Math.pow(radius, 2);
    }

}
```



```java
public class CircleTest {

    @Test
    public void testCircleArea() {
        assertThat(new Circle(5).area(), is(78.53981633974483));
    }

}
```



```java
public class ShapeCalculator {

    public double calculateArea(Shape shape) {
        return shape.area();
    }

}
```



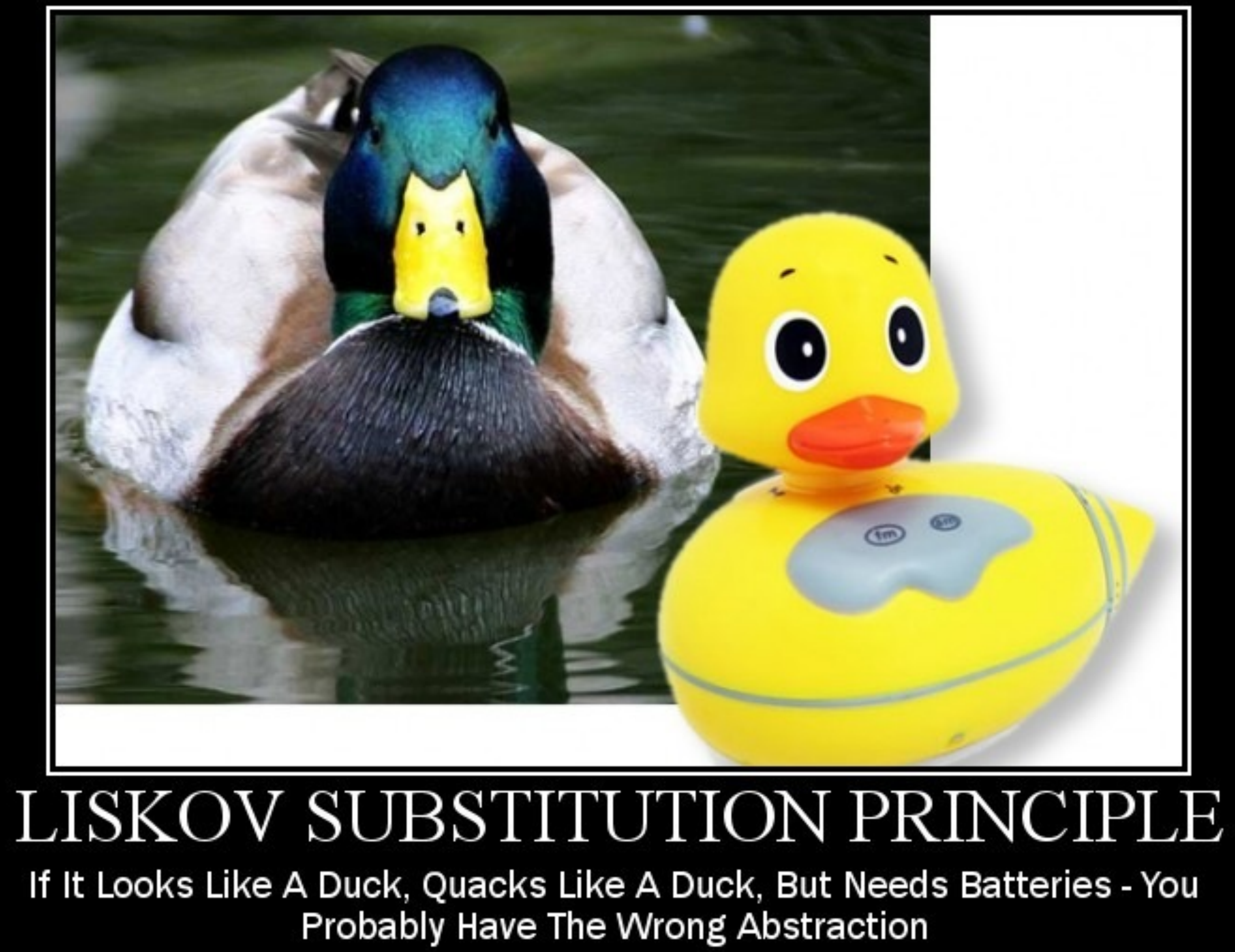
```java
public class ShapeCalculatorTest {

    @Mock
    Shape shape;

    @Before
    public void before() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testAreaOfSomeMock() {
        ShapeCalculator calculator = new ShapeCalculator();
        when(shape.area()).thenReturn(10.0);
        assertThat(calculator.calculateArea(shape), is(10.0));
        verify(shape).area();
    }

    @Test
    public void testAreaOfSomeDerivedShape() {
        ShapeCalculator calculator = new ShapeCalculator();
        assertThat(calculator.calculateArea(new Shape() {
            public double area() {
                return 10;
            }
        }), is(10.0));
    }

}
```

APPLYING OCP TO OUR TESTING

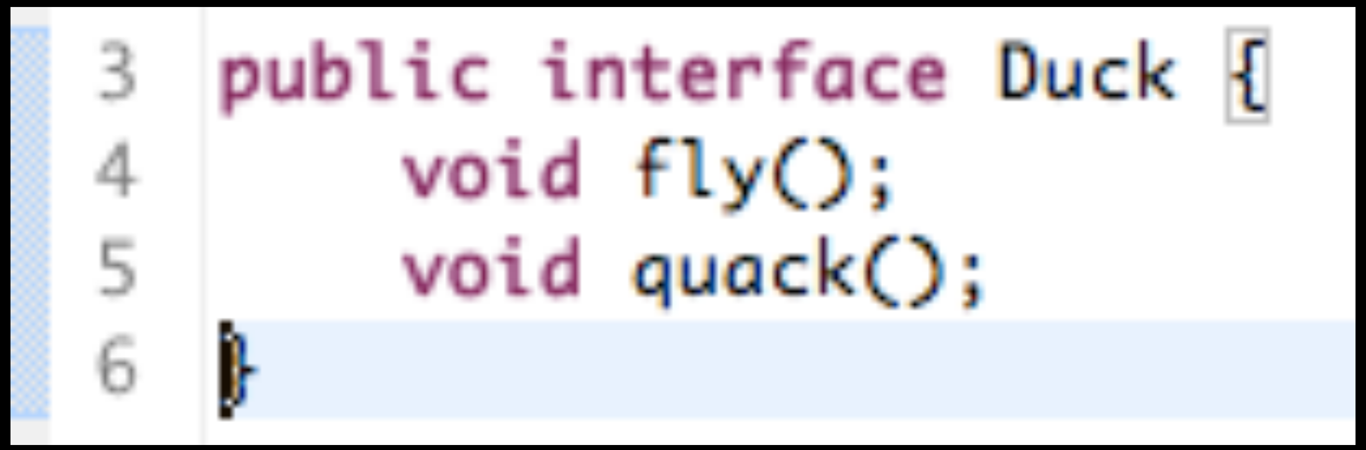
- Classes under test are open so we can:
  - inject mocked dependencies
  - override methods that make calls to external services

DERIVED CLASSES MUST BE SUBSTITUTABLE FOR THEIR BASE CLASSES.

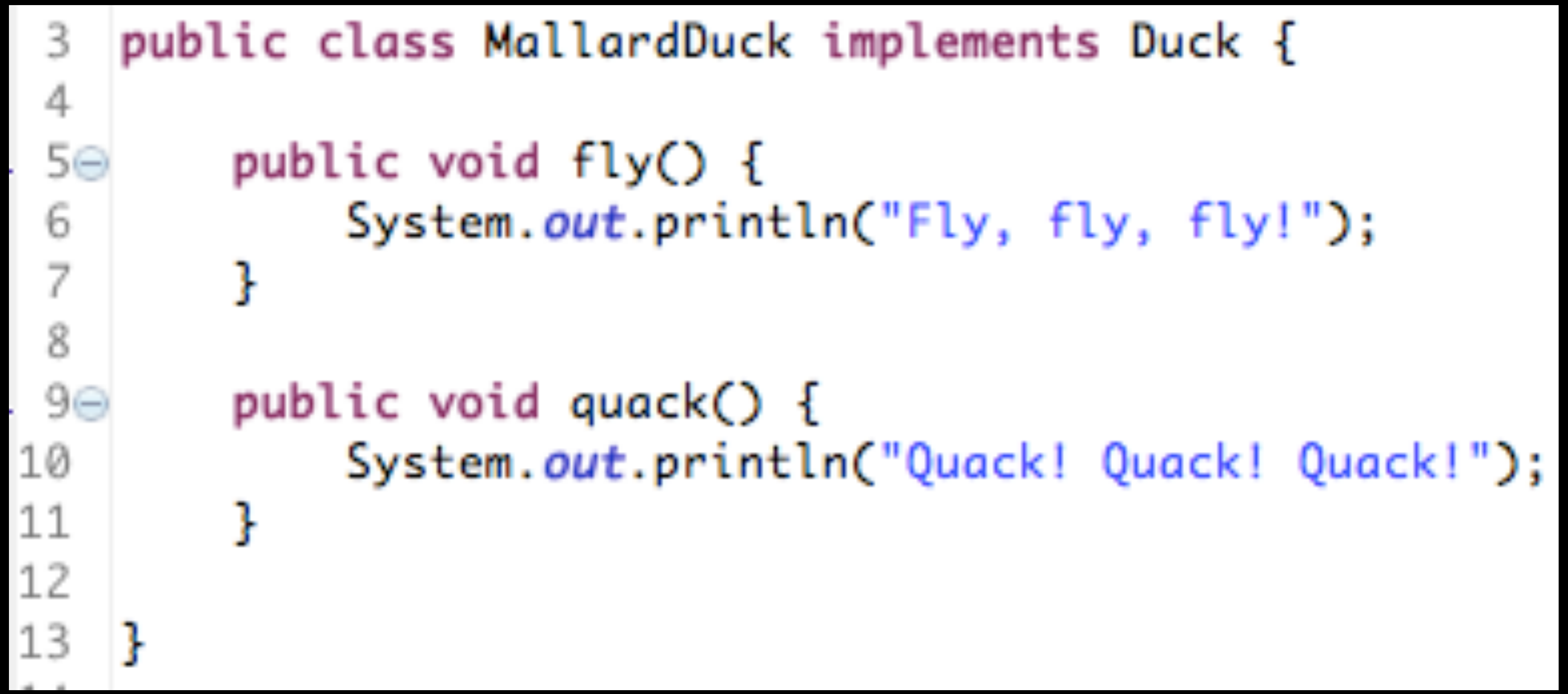
LISKOV SUBSTITUTION PRINCIPLE


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

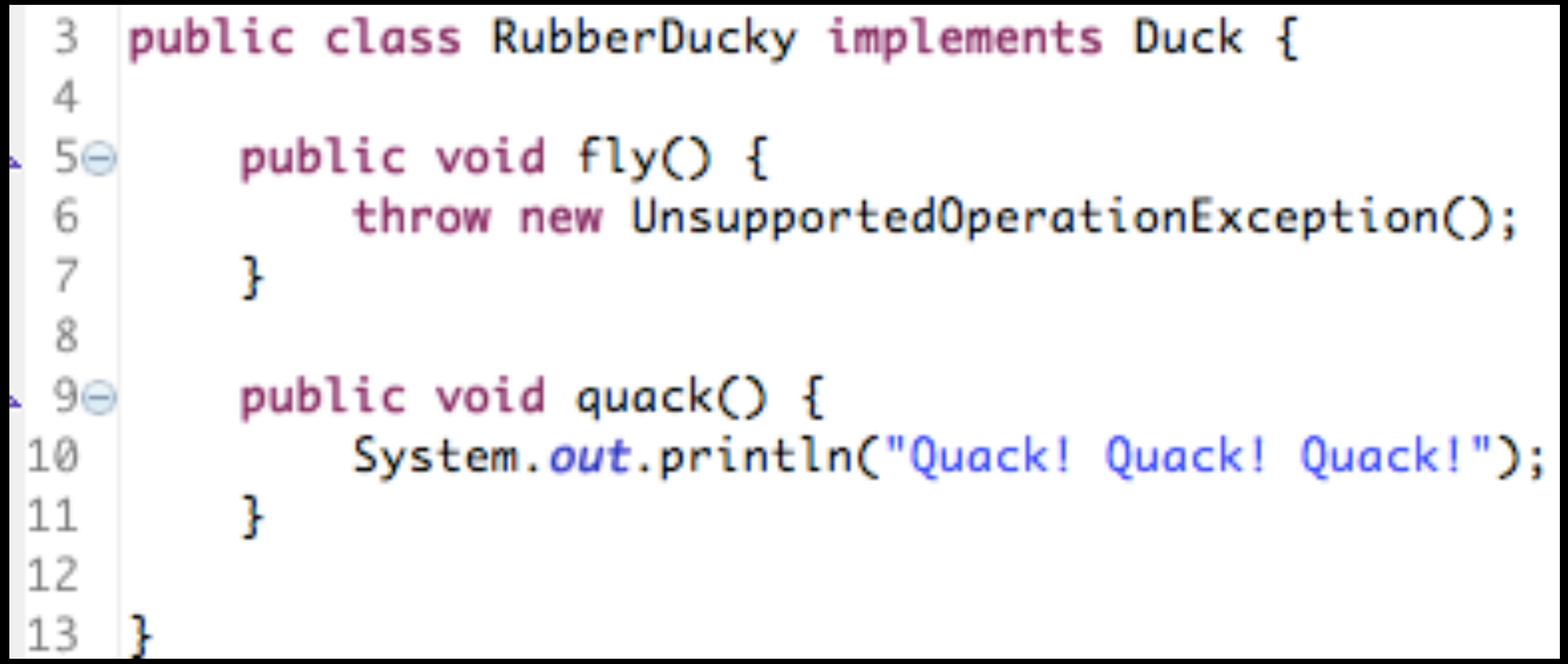
BEFORE LSP



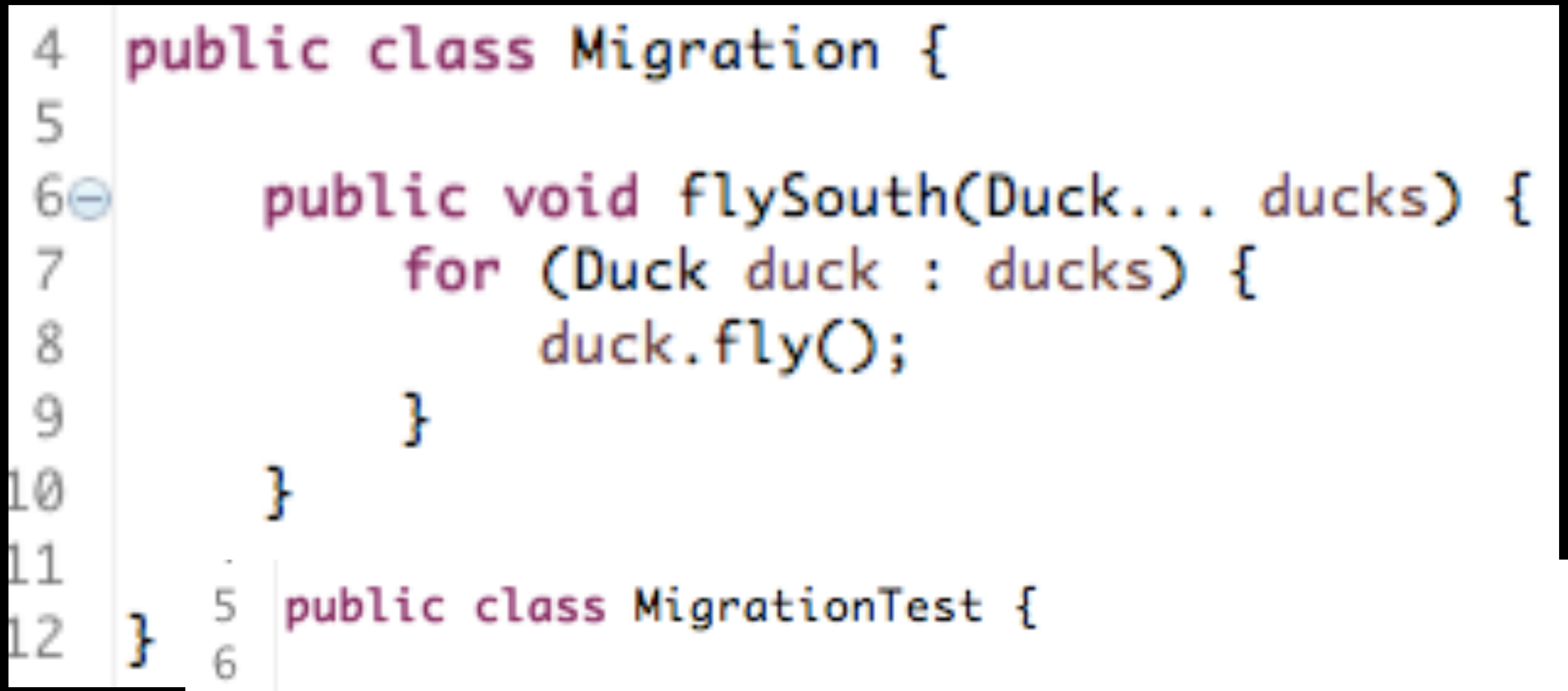
```java
public interface Duck {
    void fly();
    void quack();
}
```



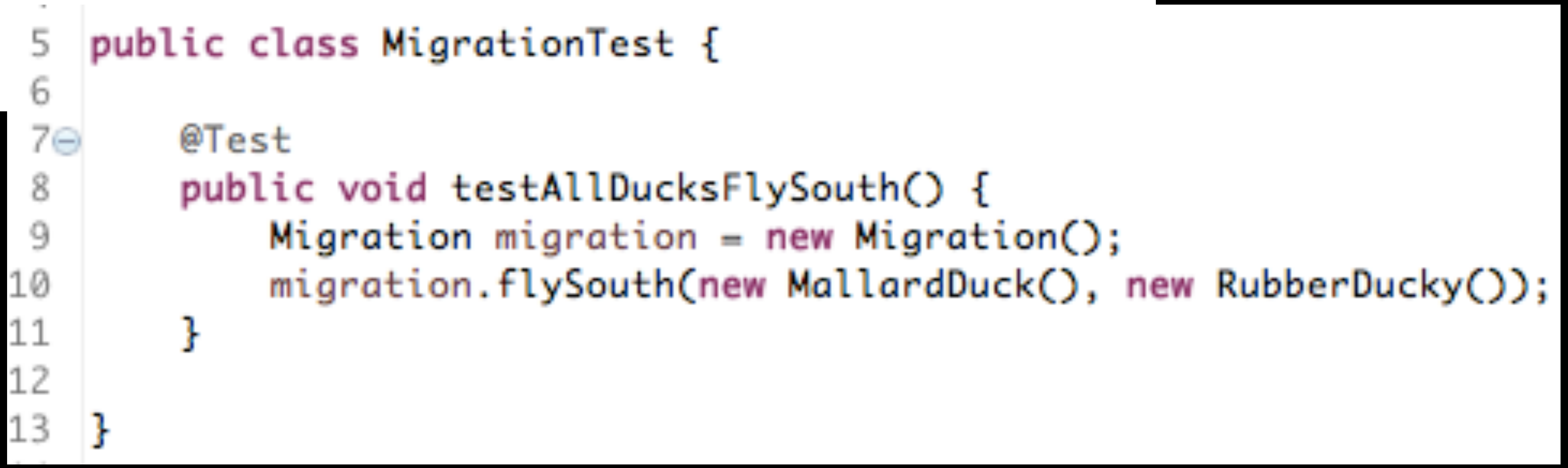
```java
public class MallardDuck implements Duck {

    public void fly() {
        System.out.println("Fly, fly, fly!");
    }

    public void quack() {
        System.out.println("Quack! Quack! Quack!");
    }

}
```



```java
public class RubberDucky implements Duck {

    public void fly() {
        throw new UnsupportedOperationException();
    }

    public void quack() {
        System.out.println("Quack! Quack! Quack!");
    }

}
```

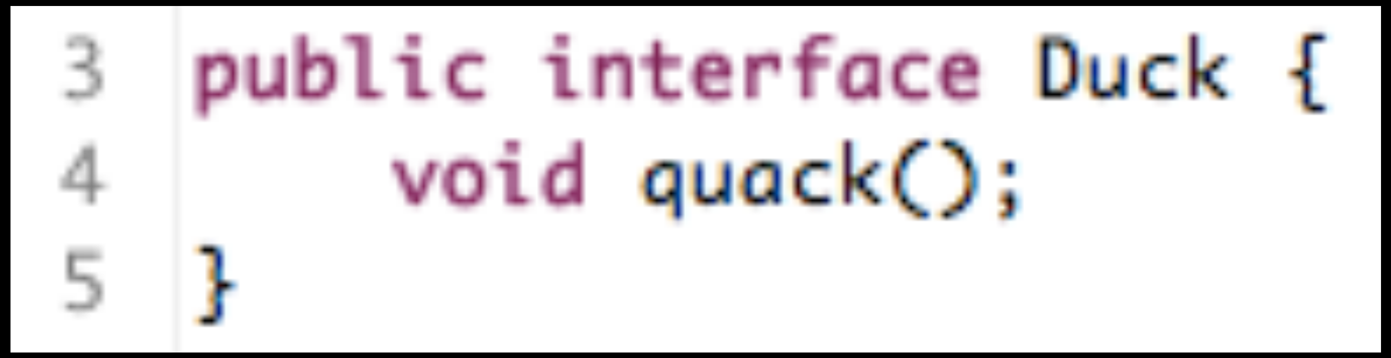


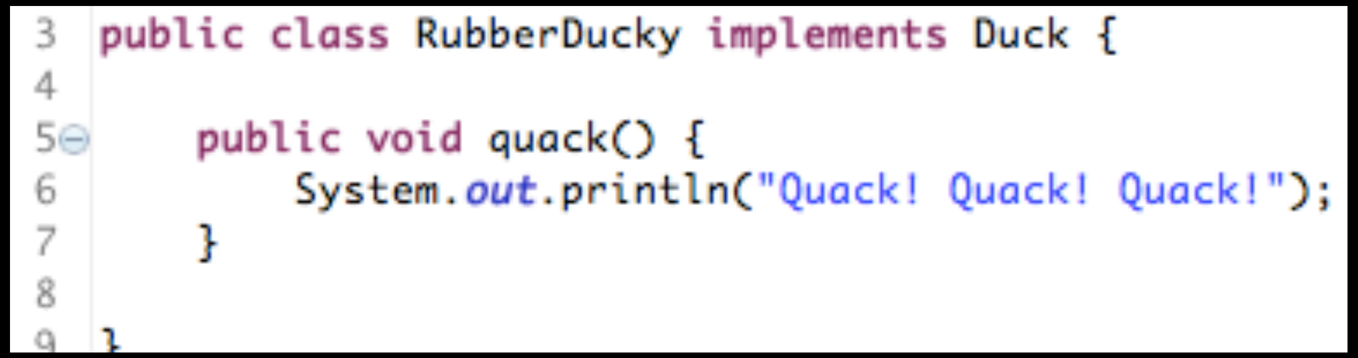
```java
public class Migration {

    public void flySouth(Duck... ducks) {
        for (Duck duck : ducks) {
            duck.fly();
        }
    }

}
```

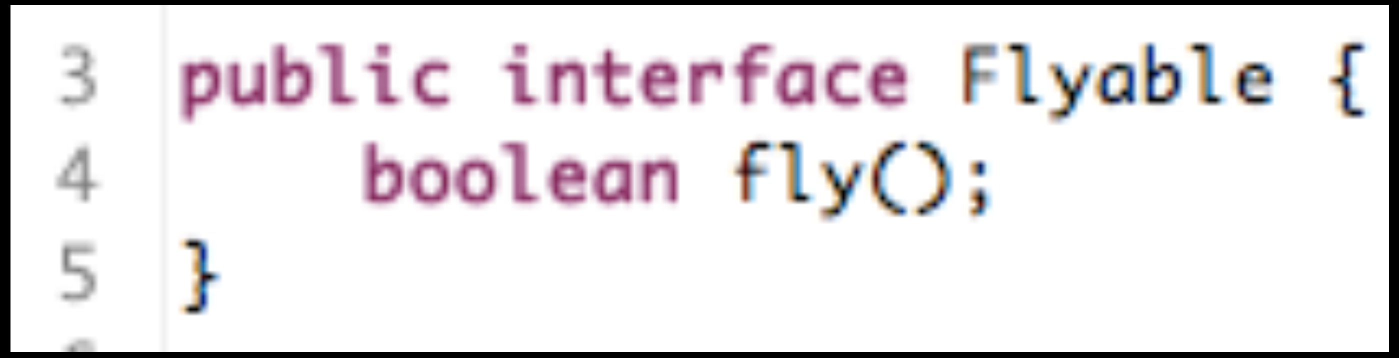


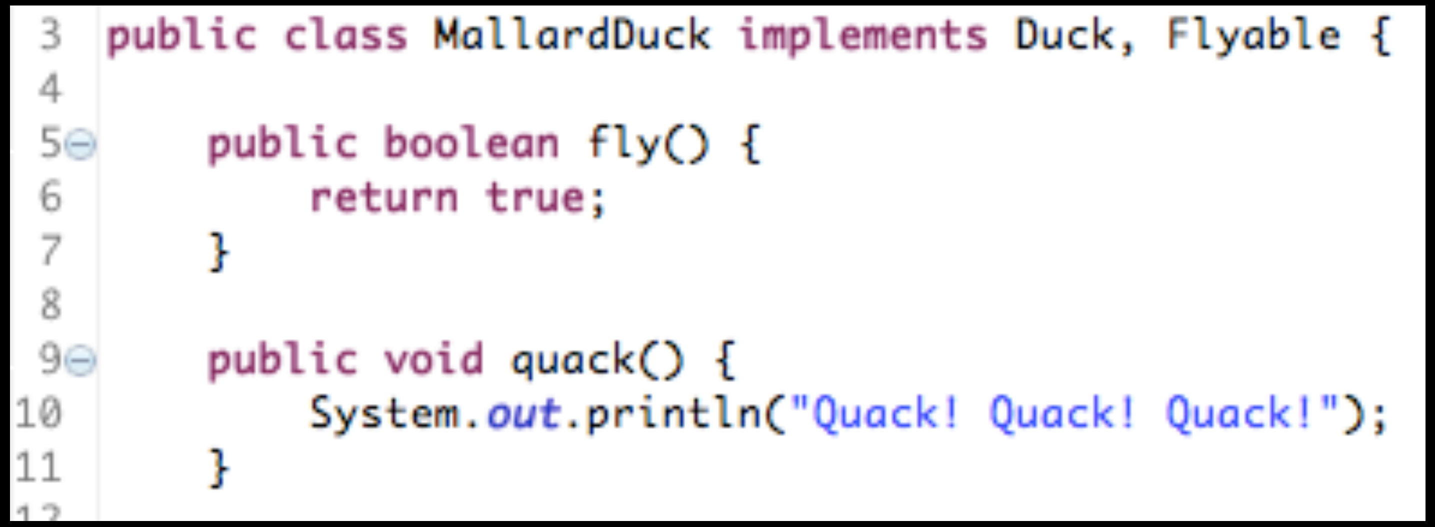
```java
public class MigrationTest {

    @Test
    public void testAllDucksFlySouth() {
        Migration migration = new Migration();
        migration.flySouth(new MallardDuck(), new RubberDucky());
    }

}
```

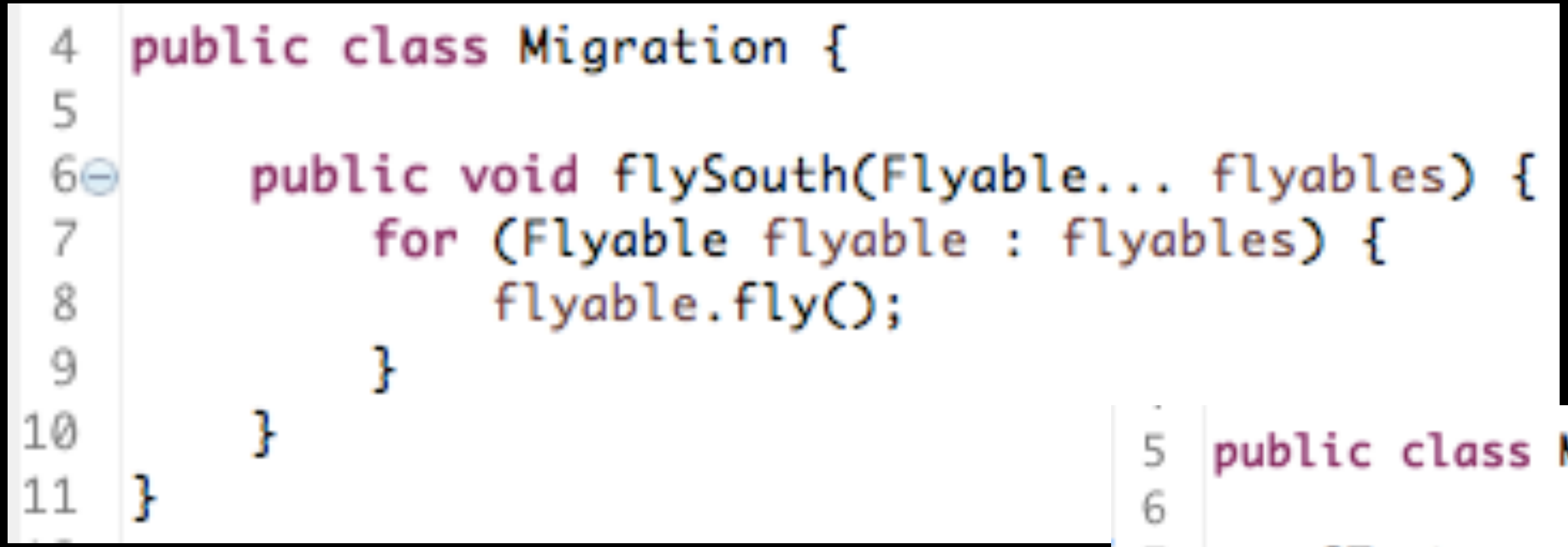
AFTER LSP

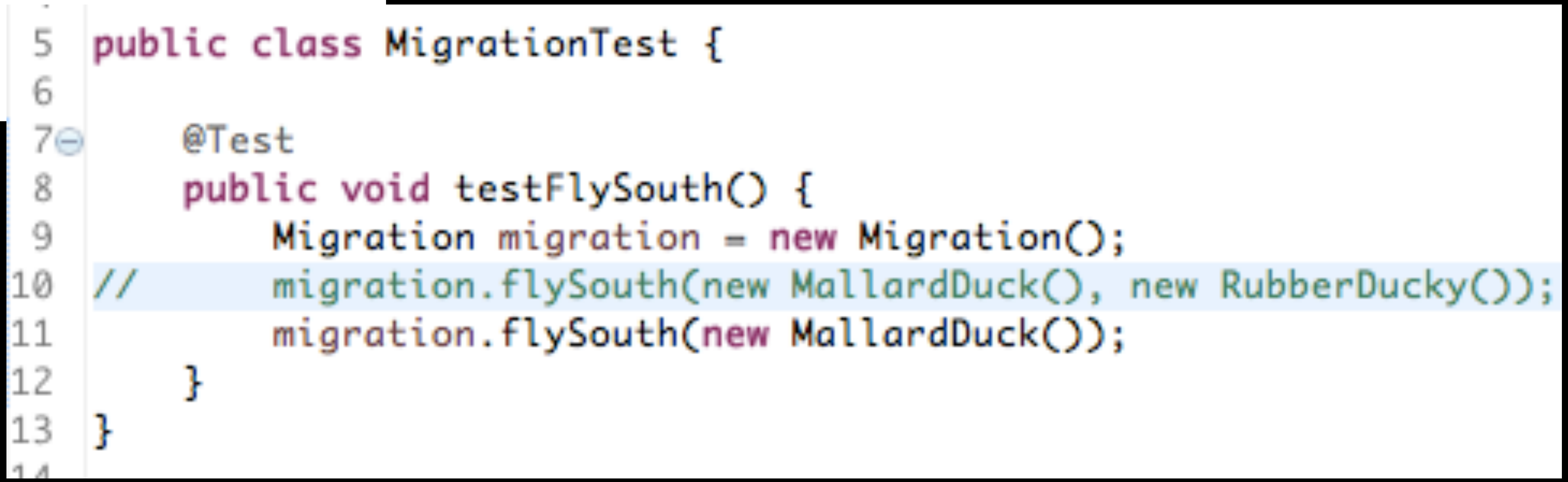

```java
public interface Duck {
    void quack();
}
```


```java
public class RubberDucky implements Duck {

    public void quack() {
        System.out.println("Quack! Quack! Quack!");
    }

}
```


```java
public interface Flyable {
    boolean fly();
}
```


```java
public class MallardDuck implements Duck, Flyable {

    public boolean fly() {
        return true;
    }

    public void quack() {
        System.out.println("Quack! Quack! Quack!");
    }
```


```java
public class Migration {

    public void flySouth(Flyable... flyables) {
        for (Flyable flyable : flyables) {
            flyable.fly();
        }
    }
}
```


```java
public class MigrationTest {

    @Test
    public void testFlySouth() {
        Migration migration = new Migration();
//      migration.flySouth(new MallardDuck(), new RubberDucky());
        migration.flySouth(new MallardDuck());
    }
}
```

APPLYING LSP TO OUR TESTING

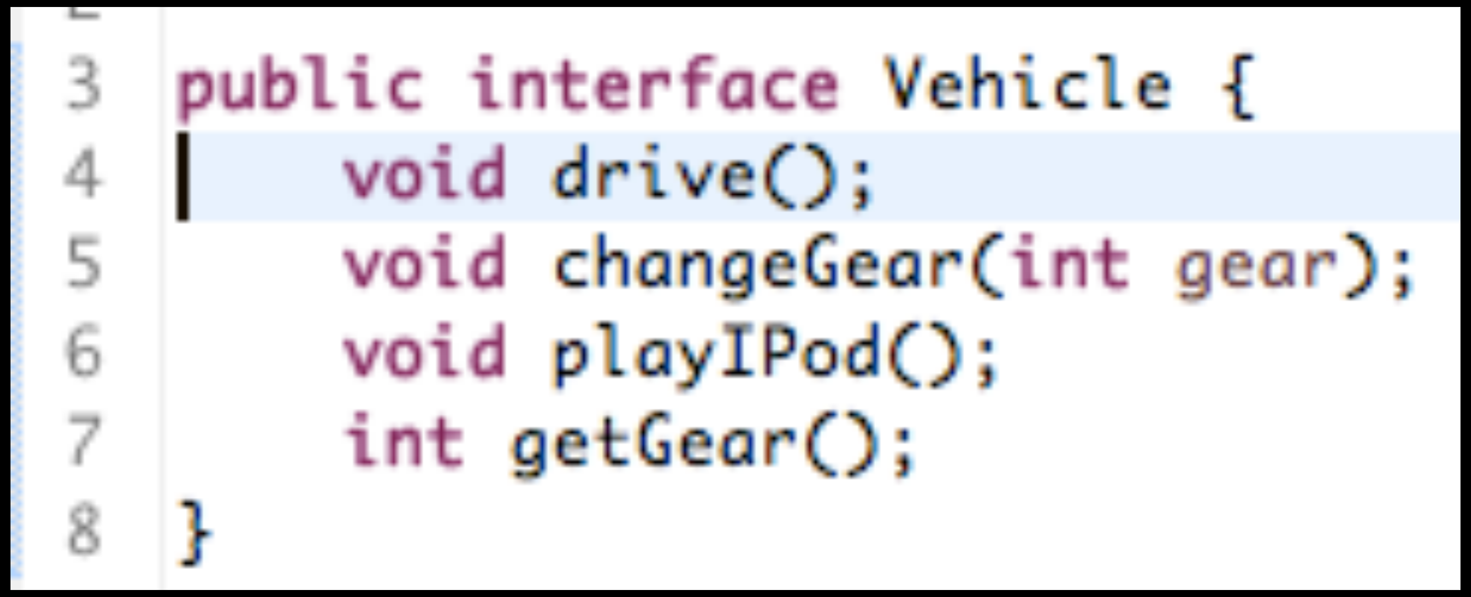
- Allows us to mock dependencies by creating substitutes (derived classes or mocks)
- Helps keep tests isolated so we only test one responsibility
- Less coupling to dependencies means we don’t need to know as much about the internals to write a test

MAKE FIND GRAINED INTERFACES THAT ARE CLIENT SPECIFIC.

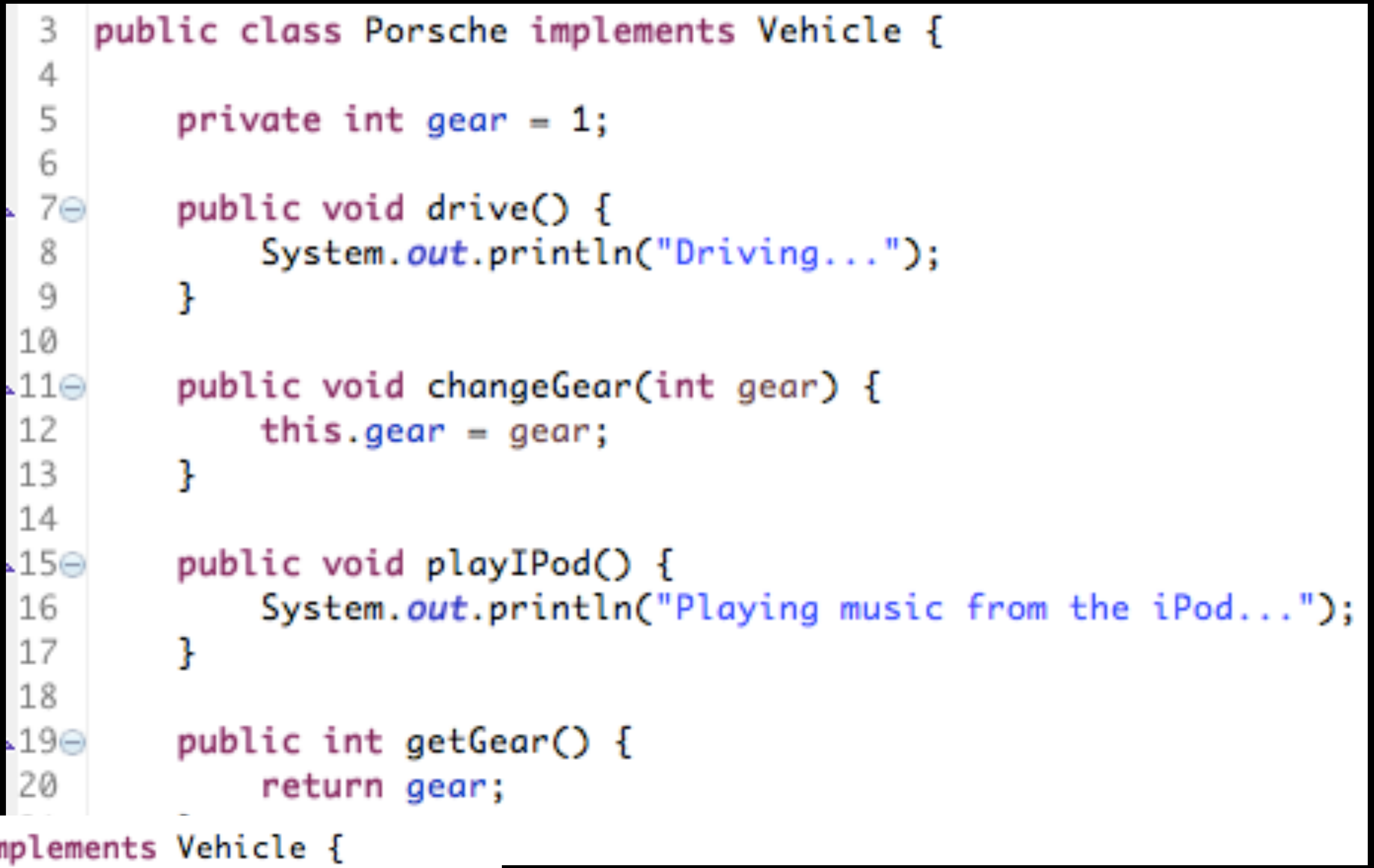
INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

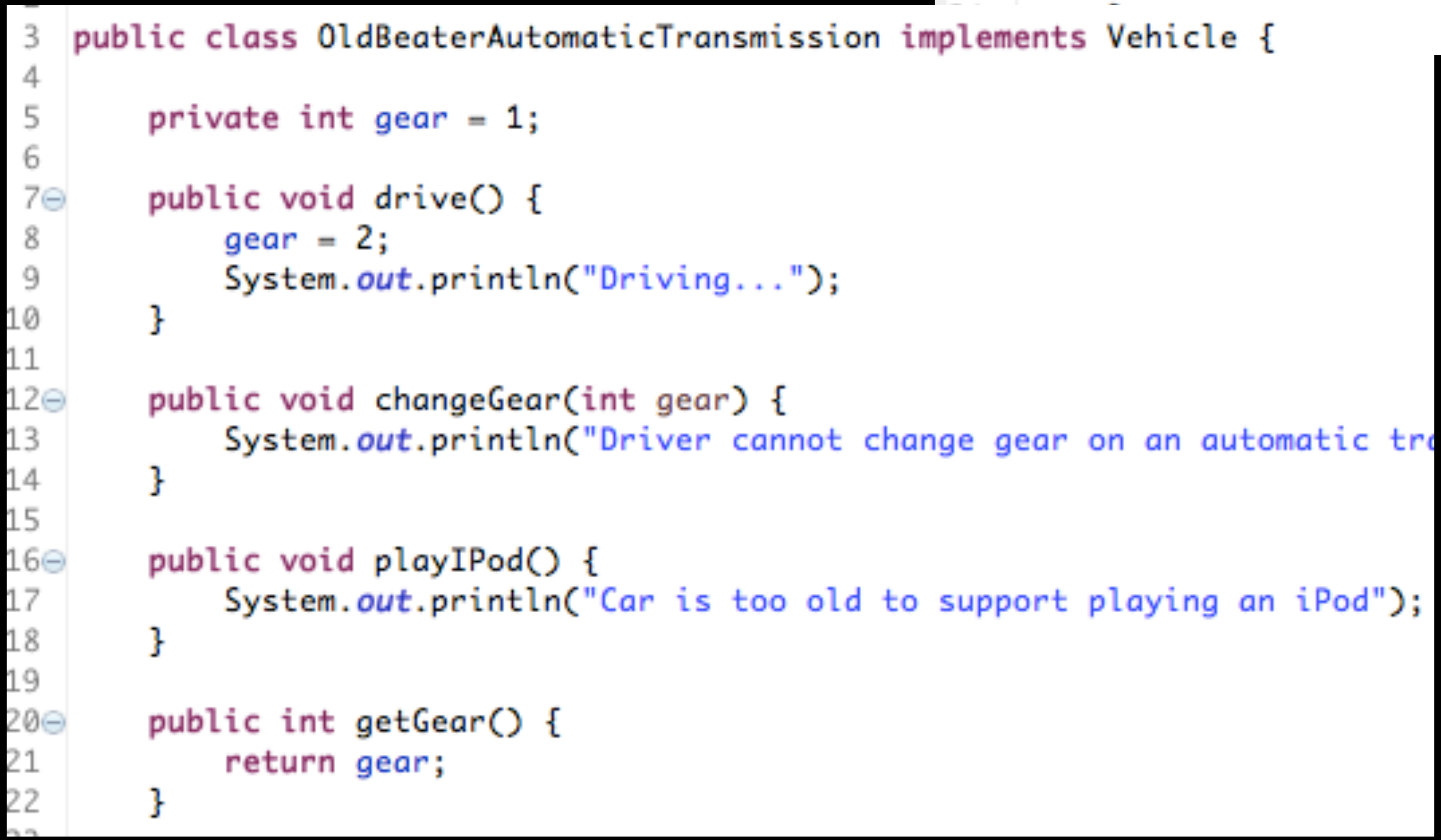
BEFORE ISP



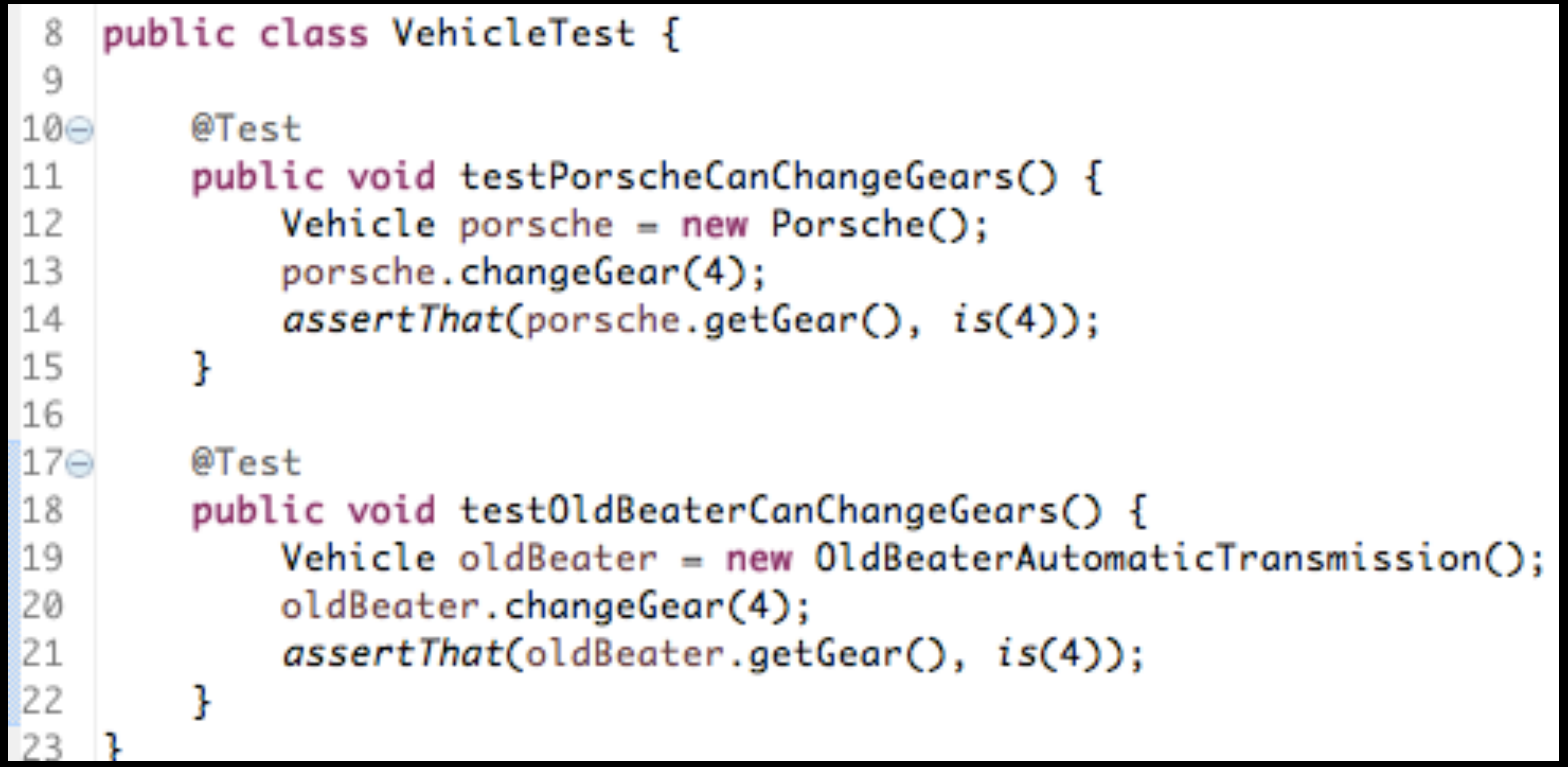
```java
public interface Vehicle {
    void drive();
    void changeGear(int gear);
    void playIPod();
    int getGear();
}
```



```java
public class Porsche implements Vehicle {

    private int gear = 1;

    public void drive() {
        System.out.println("Driving...");
    }

    public void changeGear(int gear) {
        this.gear = gear;
    }

    public void playIPod() {
        System.out.println("Playing music from the iPod...");
    }

    public int getGear() {
        return gear;
```

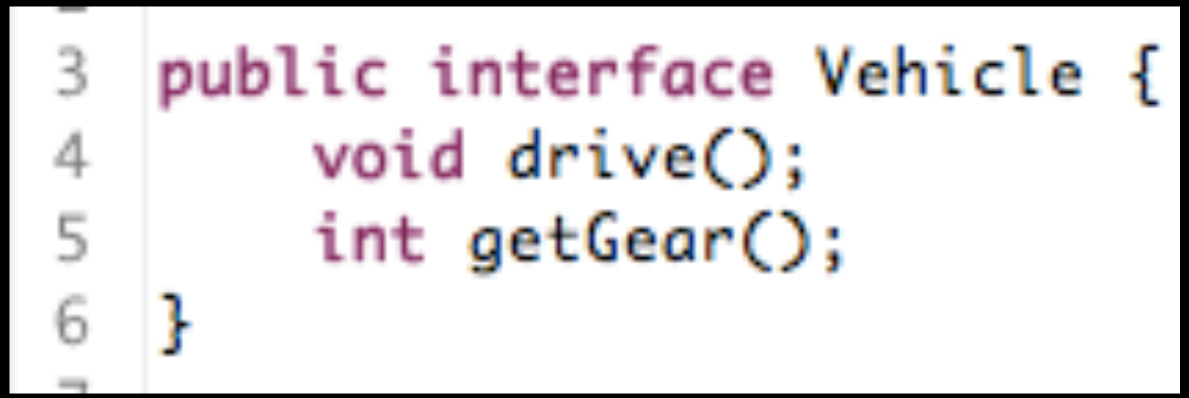


```java
public class OldBeaterAutomaticTransmission implements Vehicle {

    private int gear = 1;

    public void drive() {
        gear = 2;
        System.out.println("Driving...");
    }

    public void changeGear(int gear) {
        System.out.println("Driver cannot change gear on an automatic tr
    }

    public void playIPod() {
        System.out.println("Car is too old to support playing an iPod");
    }

    public int getGear() {
        return gear;
    }
```

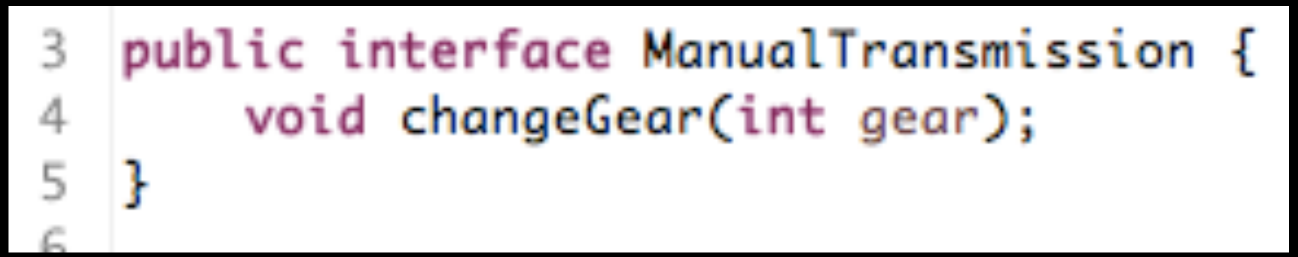


```java
public class VehicleTest {

    @Test
    public void testPorscheCanChangeGears() {
        Vehicle porsche = new Porsche();
        porsche.changeGear(4);
        assertThat(porsche.getGear(), is(4));
    }

    @Test
    public void testOldBeaterCanChangeGears() {
        Vehicle oldBeater = new OldBeaterAutomaticTransmission();
        oldBeater.changeGear(4);
        assertThat(oldBeater.getGear(), is(4));
    }
}
```

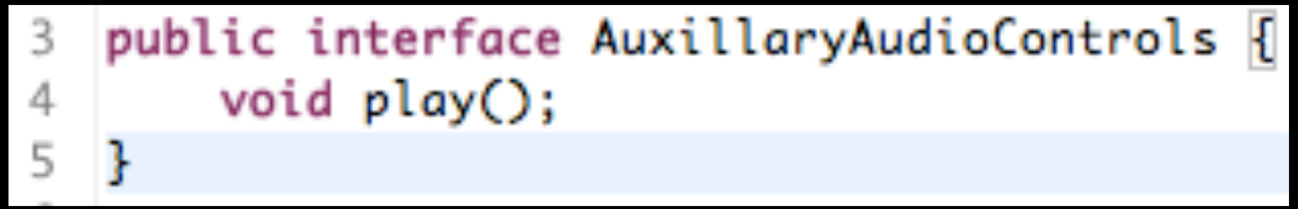
AFTER ISP



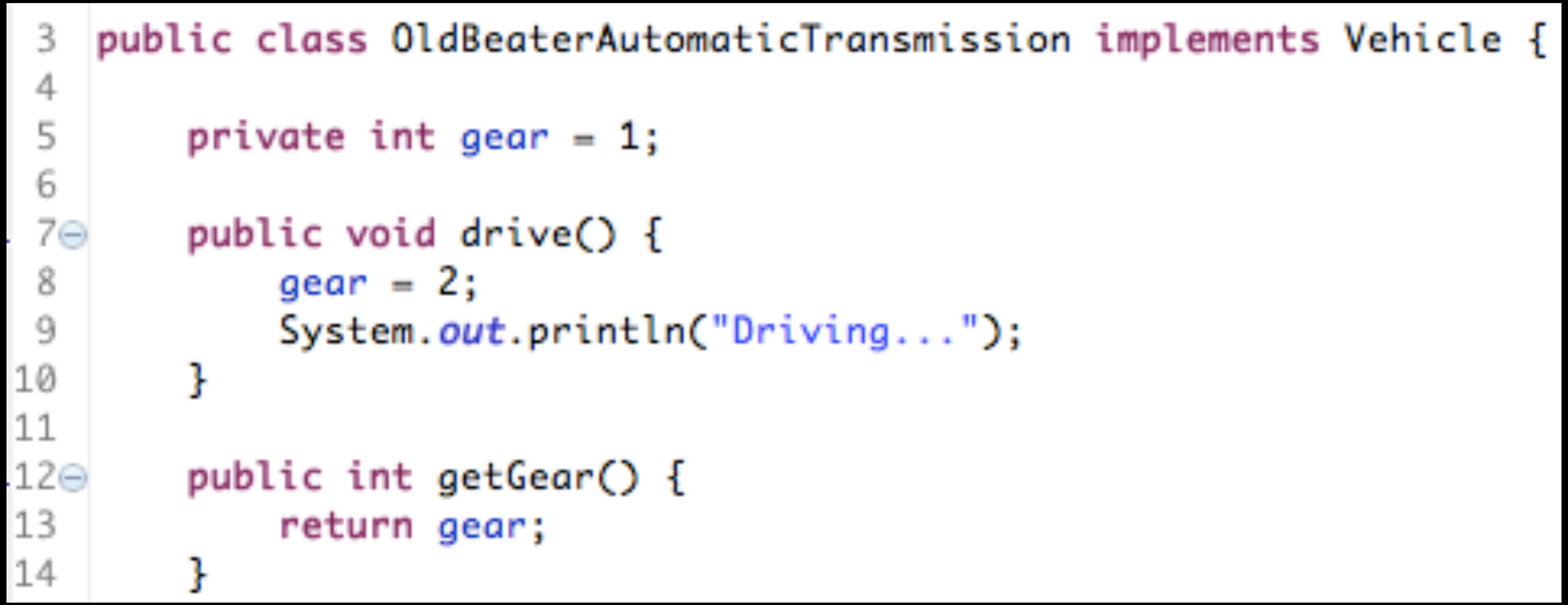
```java
public interface Vehicle {
    void drive();
    int getGear();
}
```



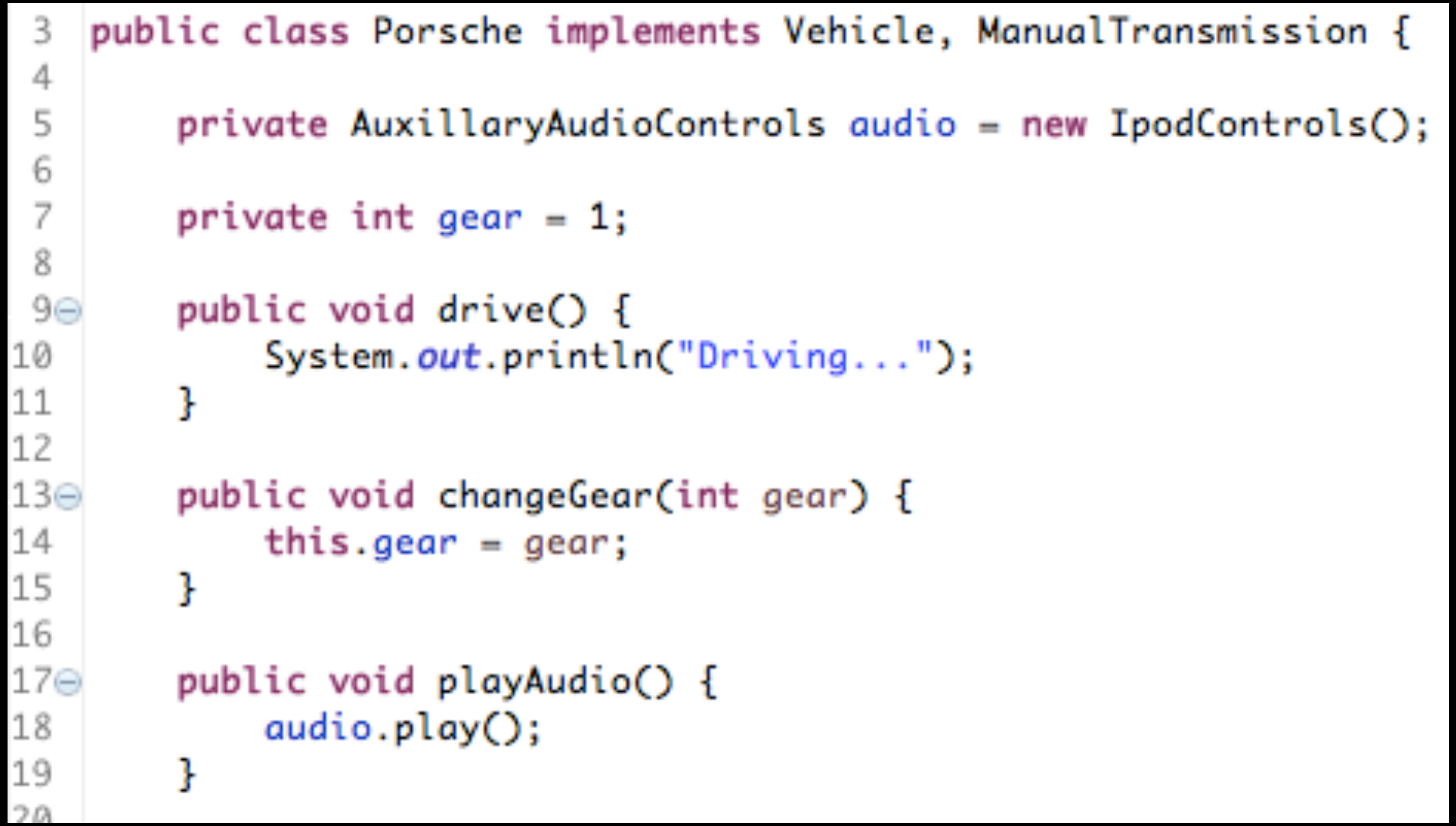
```java
public interface ManualTransmission {
    void changeGear(int gear);
}
```



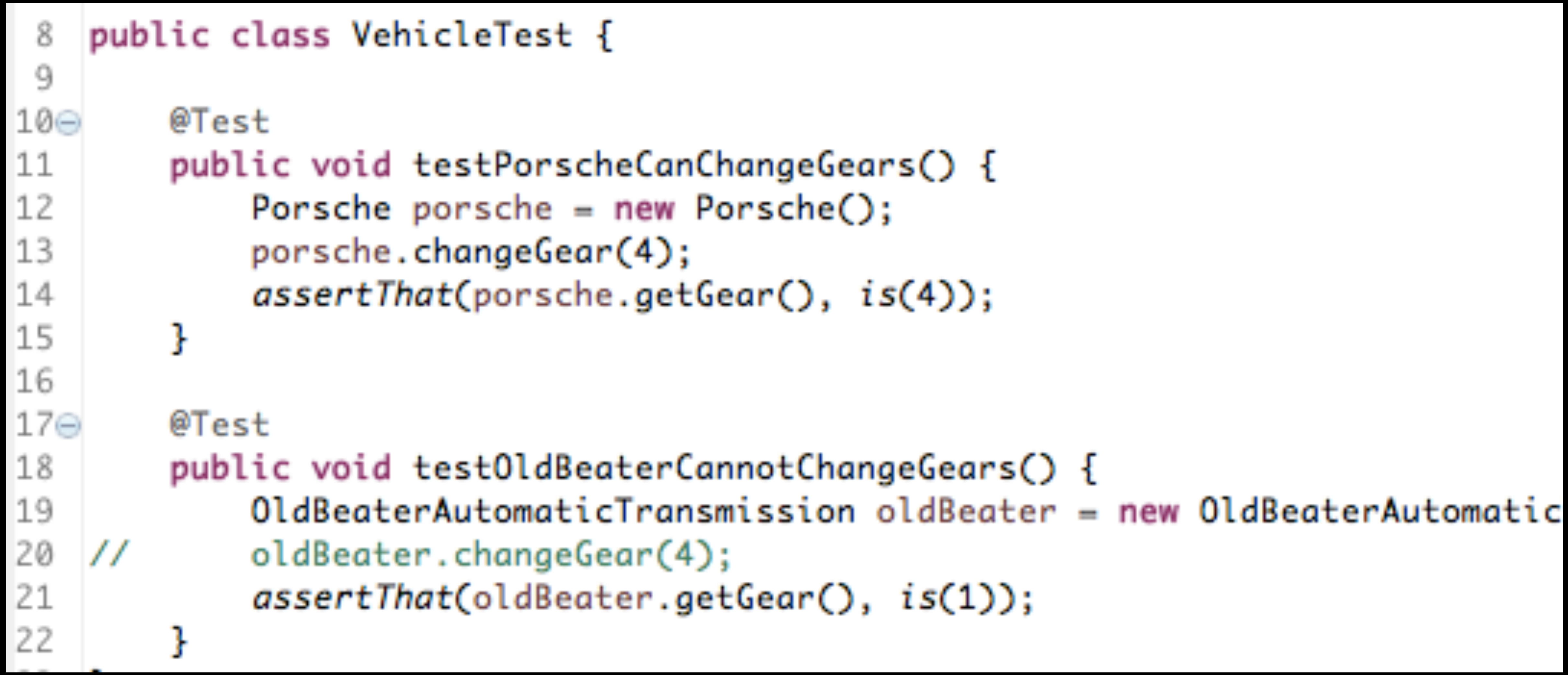
```java
public interface AuxillaryAudioControls {
    void play();
}
```



```java
public class OldBeaterAutomaticTransmission implements Vehicle {

    private int gear = 1;

    public void drive() {
        gear = 2;
        System.out.println("Driving...");
    }

    public int getGear() {
        return gear;
    }
```



```java
public class Porsche implements Vehicle, ManualTransmission {

    private AuxillaryAudioControls audio = new IpodControls();

    private int gear = 1;

    public void drive() {
        System.out.println("Driving...");
    }

    public void changeGear(int gear) {
        this.gear = gear;
    }

    public void playAudio() {
        audio.play();
    }
```



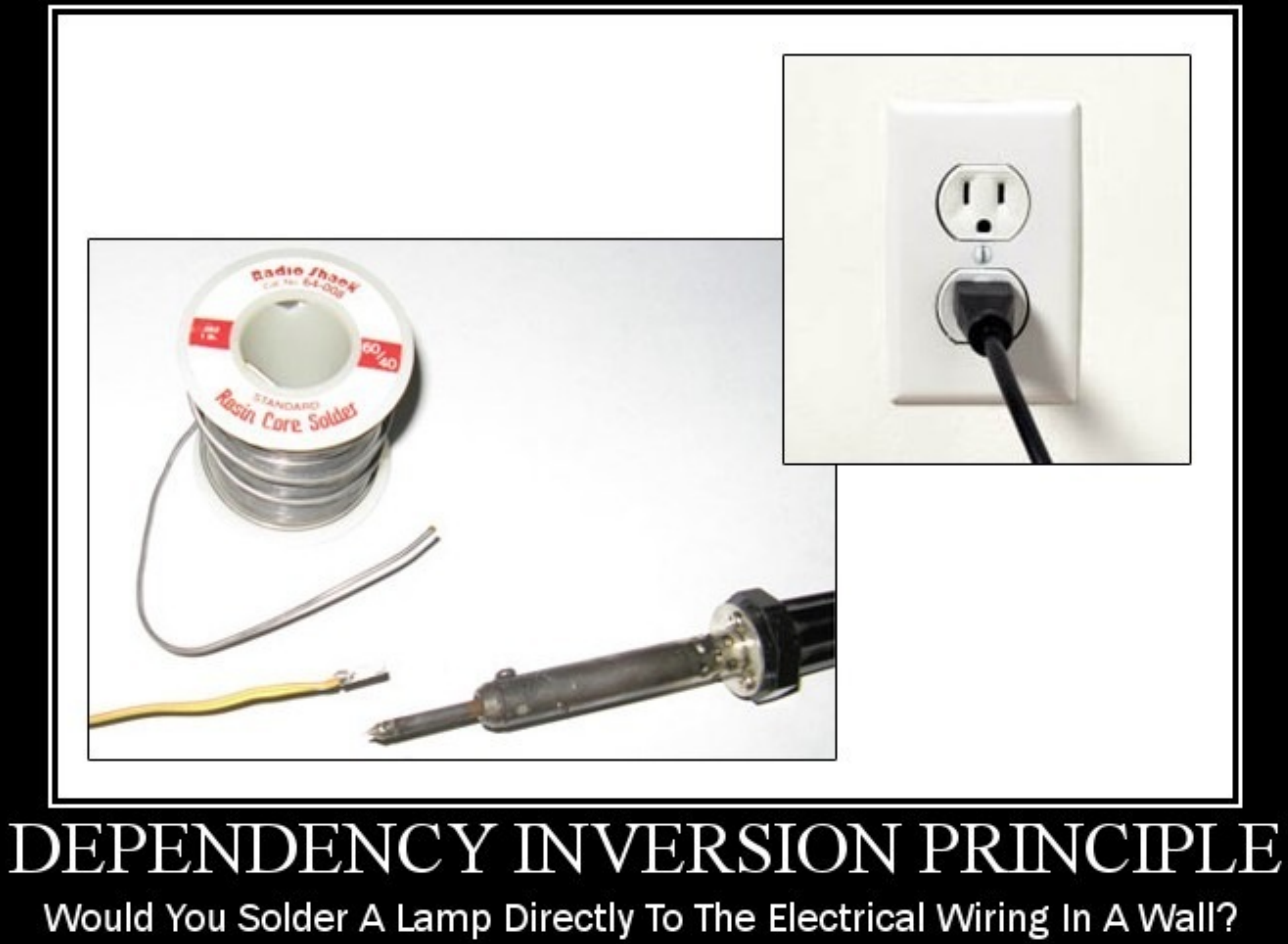
```java
public class VehicleTest {

    @Test
    public void testPorscheCanChangeGears() {
        Porsche porsche = new Porsche();
        porsche.changeGear(4);
        assertThat(porsche.getGear(), is(4));
    }

    @Test
    public void testOldBeaterCannotChangeGears() {
        OldBeaterAutomaticTransmission oldBeater = new OldBeaterAutomatic
//      oldBeater.changeGear(4);
        assertThat(oldBeater.getGear(), is(1));
    }
```

APPLYING ISP TO OUR TESTING

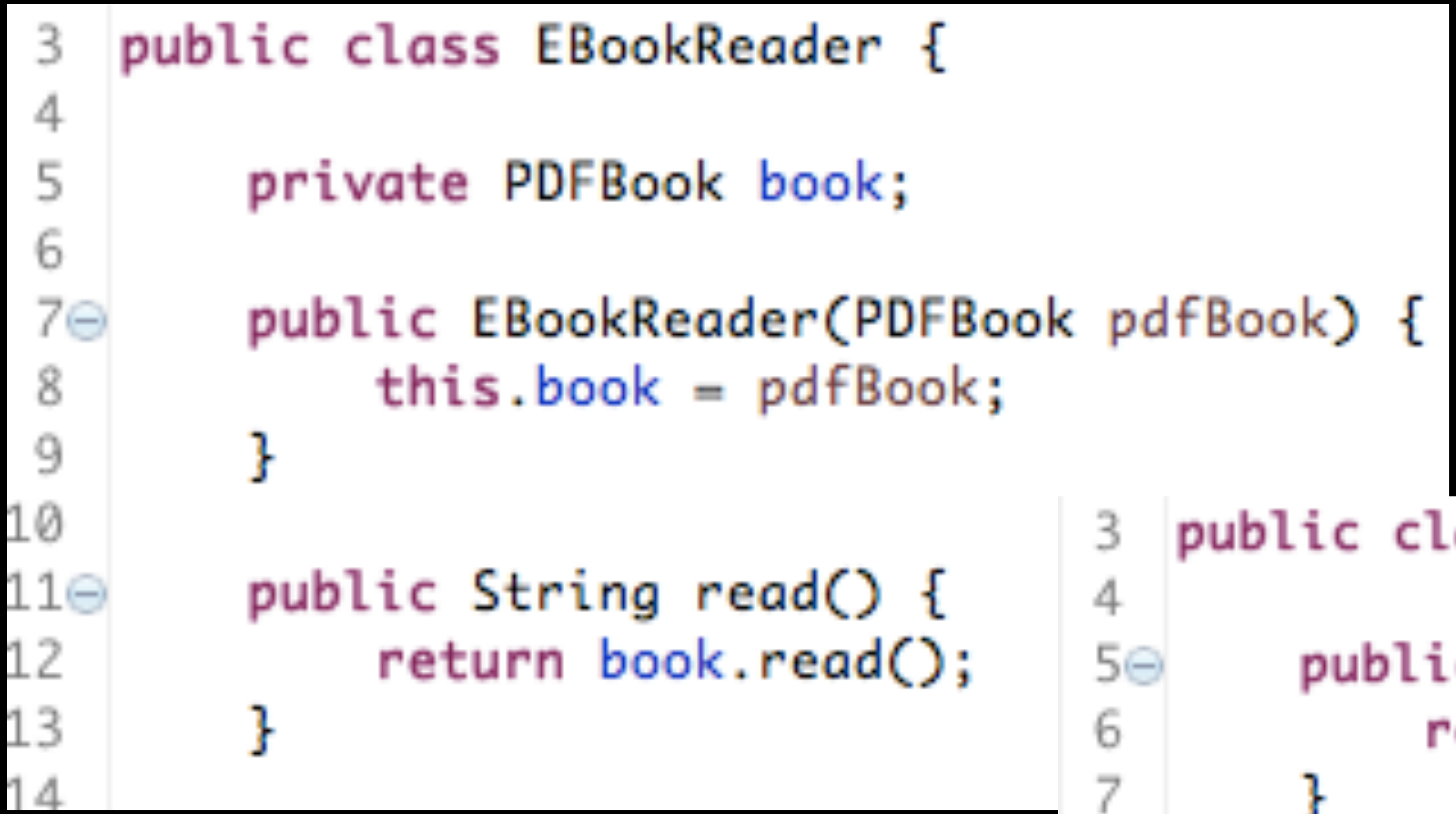
- Classes depend on narrower interfaces, making
  - tests smaller
  - mocking easier

DEPEND ON ABSTRACTIONS, NOT ON CONCRETIONS.

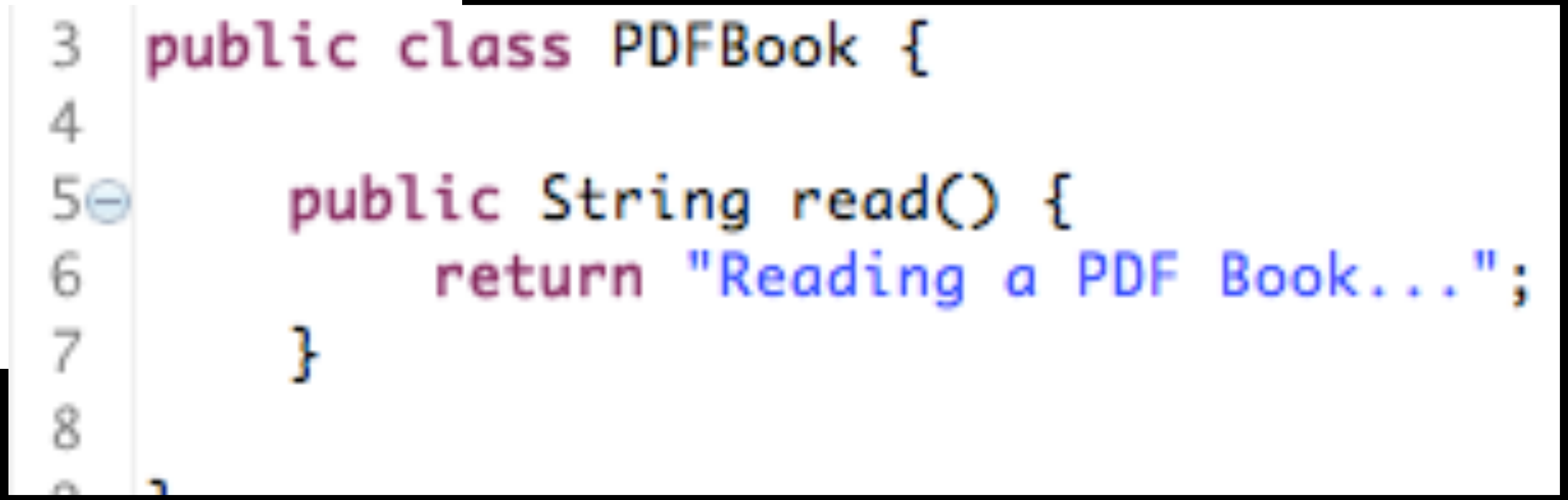
DEPENDENCY INVERSION PRINCIPLE


Radio Shack
Rosin Core Solder
60/40
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

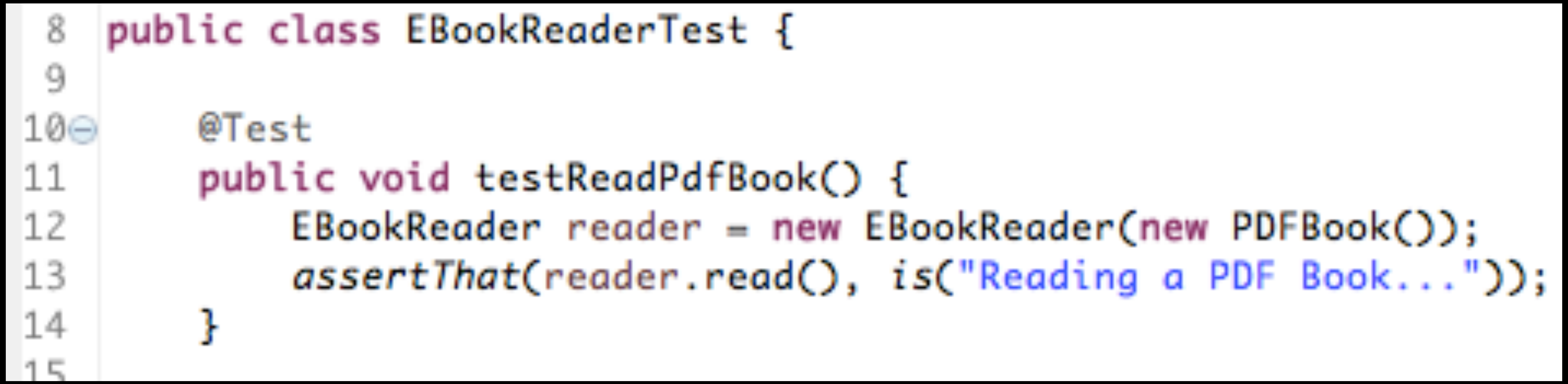
BEFORE DIP



```java
public class EBookReader {

    private PDFBook book;

    public EBookReader(PDFBook pdfBook) {
        this.book = pdfBook;
    }

    public String read() {
        return book.read();
    }
```

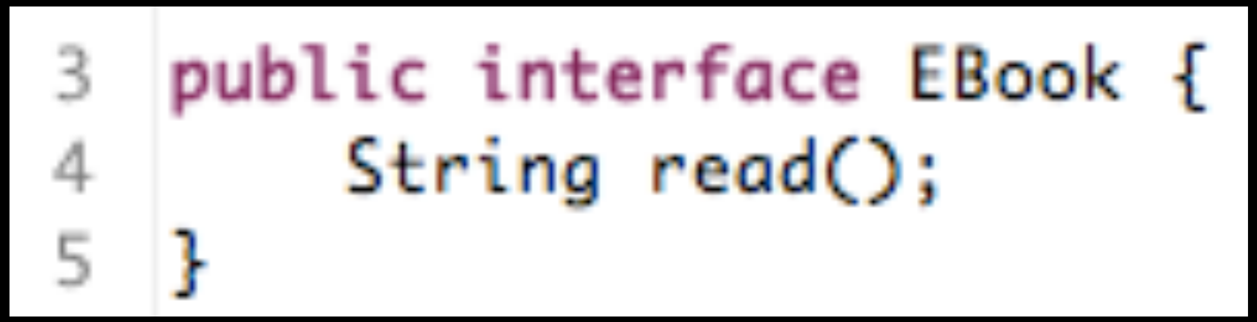


```java
public class PDFBook {

    public String read() {
        return "Reading a PDF Book...";
    }

}
```

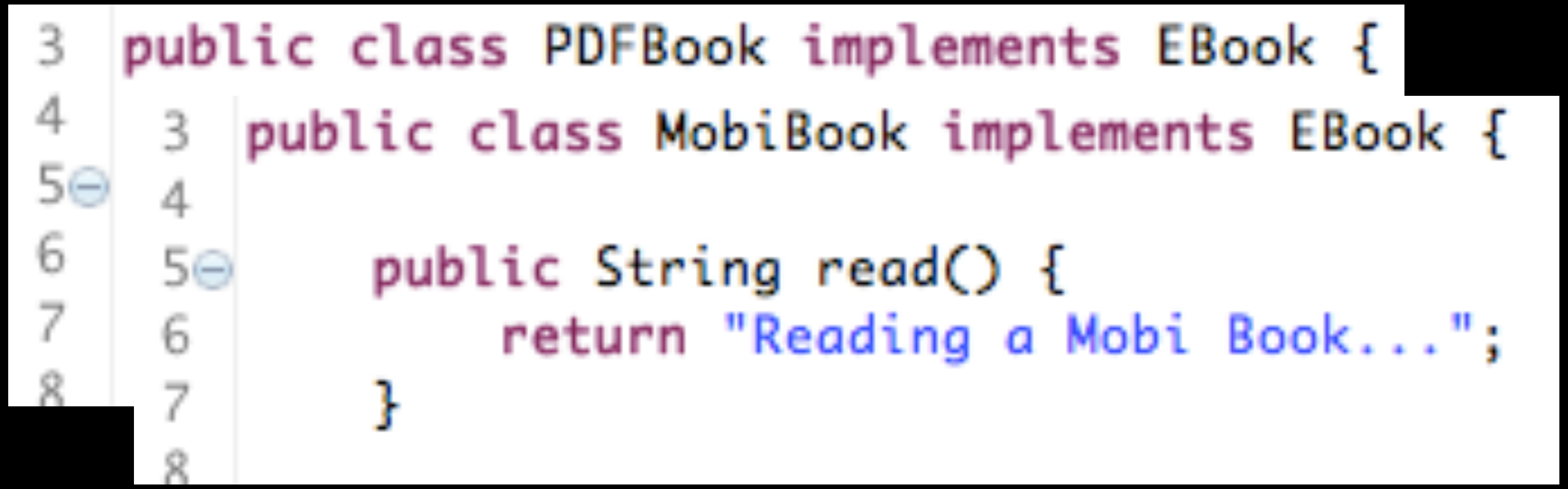


```java
public class EBookReaderTest {

    @Test
    public void testReadPdfBook() {
        EBookReader reader = new EBookReader(new PDFBook());
        assertThat(reader.read(), is("Reading a PDF Book..."));
    }
```

AFTER DIP

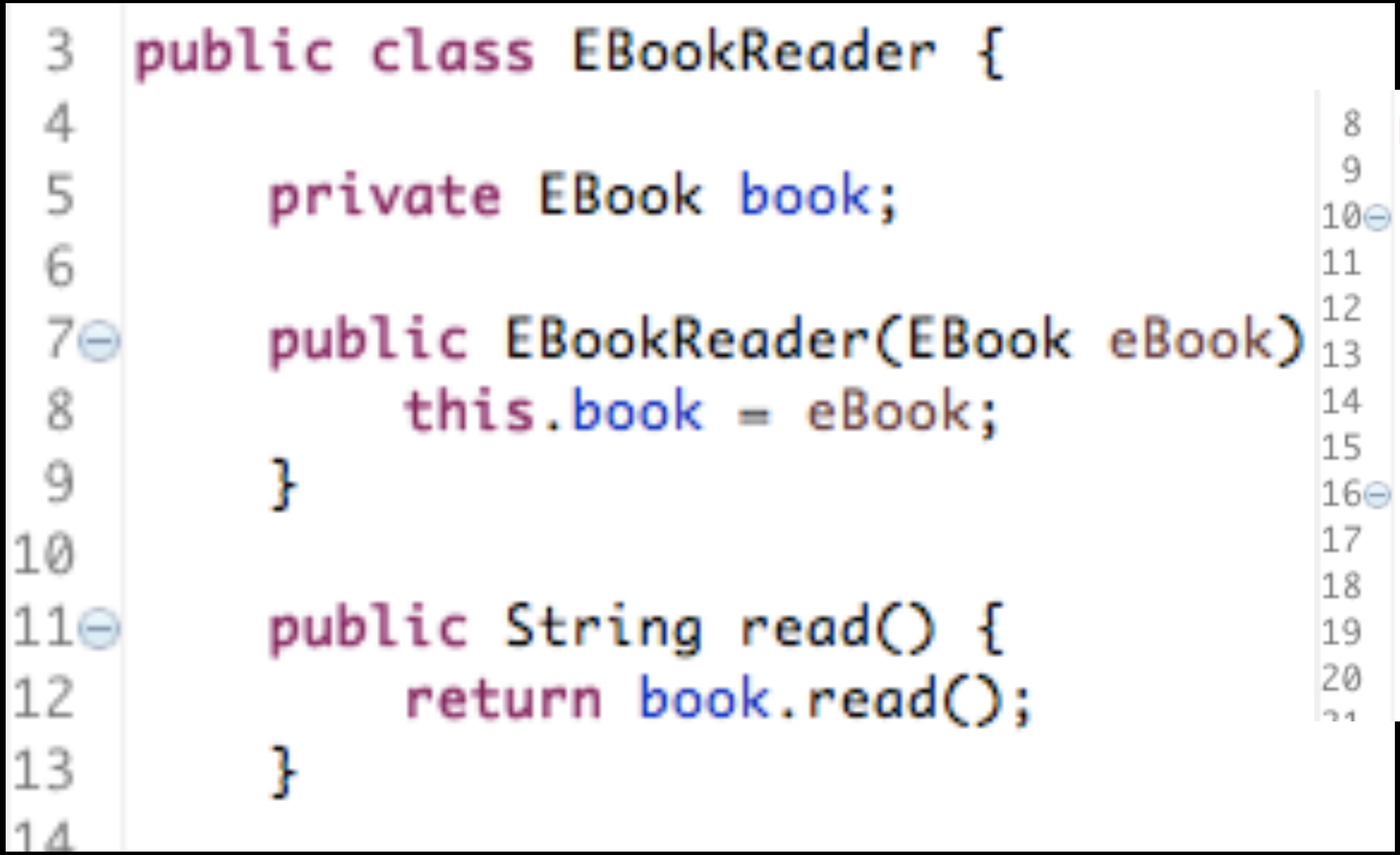


```java
public interface EBook {
    String read();
}
```

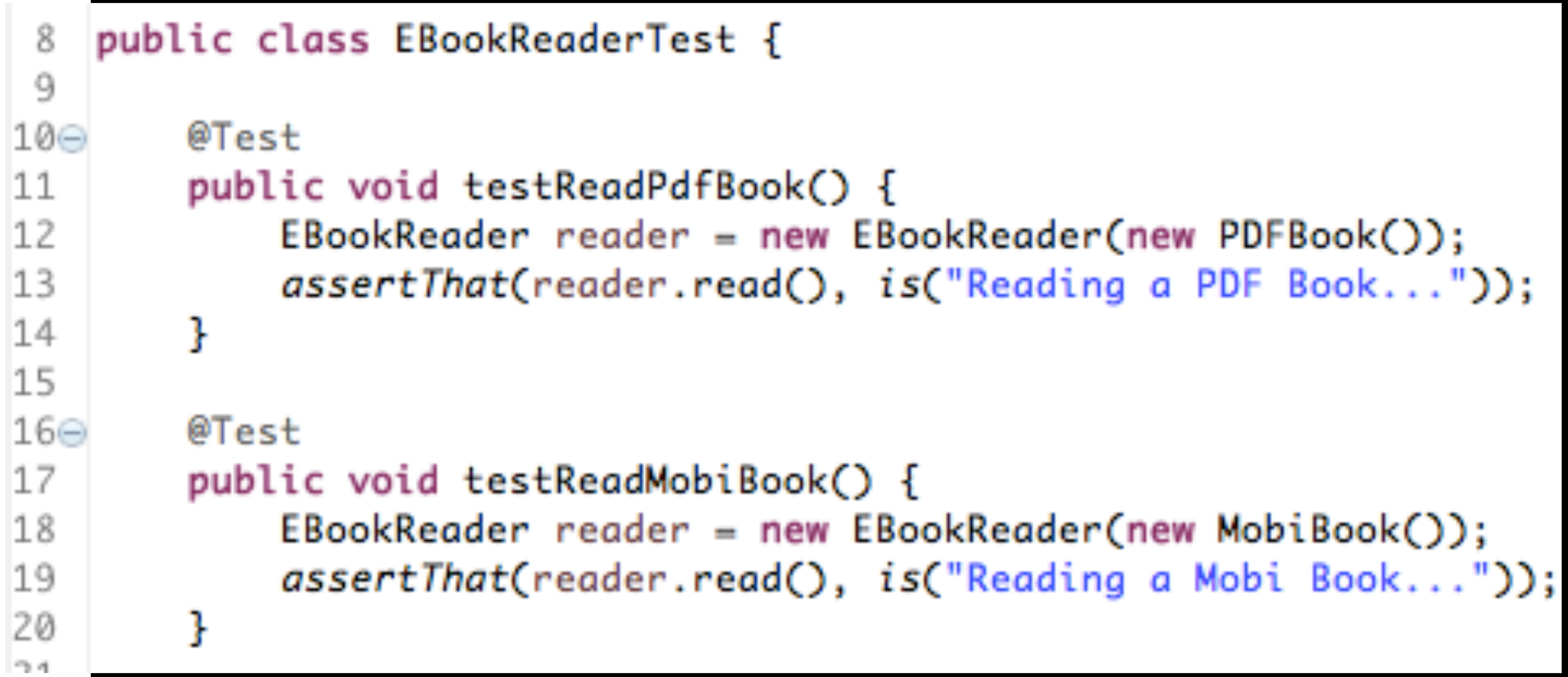


```java
public class PDFBook implements EBook {
```

```java
public class MobiBook implements EBook {

    public String read() {
        return "Reading a Mobi Book...";
    }
```



```java
public class EBookReader {

    private EBook book;

    public EBookReader(EBook eBook) {
        this.book = eBook;
    }

    public String read() {
        return book.read();
    }
```



```java
public class EBookReaderTest {

    @Test
    public void testReadPdfBook() {
        EBookReader reader = new EBookReader(new PDFBook());
        assertThat(reader.read(), is("Reading a PDF Book..."));
    }

    @Test
    public void testReadMobiBook() {
        EBookReader reader = new EBookReader(new MobiBook());
        assertThat(reader.read(), is("Reading a Mobi Book..."));
    }
```

APPLYING DIP TO OUR TESTING

- Writing more decoupled code allows mocking and injecting of dependencies
- If you skip these abstractions, your tests become more end-to-end tests because you are unable to swap out implementations for mocks.

NOW WHAT?

- Use the SOLID Principles to guide your way toward keeping your codebase clean, easier to extend, maintain, and reuse
- SOLID is not the law
- TDD does not guarantee good design
- Don’t forget to engage your brain!

QUESTIONS?

HTTPS://WWW.THATCONFERENCE.COM/SPONSORS

PLEASE THANK THE SPONSORS!

8/10/15 - 8/12/15

HTTPS://GITHUB.COM/JENNAPEDERSON/TDD-USING-THE-SOLID-PRINCIPLES

CODE & RESOURCES

SOLID MOTIVATIONAL PHOTO CREDITS: HTTP://BIT.LY/MOTIVATIONAL_SOLID

@JENNAPEDERSON
WWW.612SOFTWAREFOUNDRY.COM

THANK YOU!