

EECS2030 Fall 2022

Review Notes: Deriving Classes and Methods from JUnit Tests Expectation and Strategy

CHEN-WEI WANG

1 Expectation

– You are given a single JUnit test class, where the test methods collectively illustrate:

- How objects are instantiated from **certain classes**

e.g., `Person jim = new Person(78.0, 1.82);`

- How **certain mutator methods** may be called upon these objects to modify their attribute values

e.g., `jim.gainWeightBy(2.0);`

- How **certain accessor methods** may be called upon these objects to obtain values

e.g., `double jimBMI = jim.getBMI();`

- What the **expected return values** of accessor method calls should be

e.g., `assertEquals(24.15, jimBMI, 0.1);`

– Only the JUnit test class is given to you. **No other classes are given.**

– Therefore, to start with, there are lots of compilation errors, which is **expected** because none of the

- classes (e.g., `Person`),
- constructors (e.g., `Person(double weight, double height)`),
- mutators (e.g., `void gainWeightBy(double units)`), and
- accessors (e.g., `double getBMI()`)

have been declared and defined.

– Your task, then, is to **create and define all these classes and methods**, such that:

- All your Java classes and the given JUnit test class **compile**.
- Running the JUnit test class gives a **green bar** (i.e., all tests pass).

– Programming IDEs such as Eclipse are able to fix such compilation errors for you. **However, you are advised to follow the guidance below to fix these compilation errors manually, because: 1) it helps you better understand how the intended classes and methods work together; and 2) you may be tested in a written test or exam without the assistance of IDEs.**

2 A Small Example

2.1 What You Are Given: A JUnit Tester

This tester class must not be modified.

```
1 public class TestCounter {
2     @Test
3     public void test_Counter() {
4         Counter c1 = new Counter();
5         Counter c2 = new Counter(5);
6         int c1Value = c1.getValue();
7         int c2Value = c2.getValue();
8         assertEquals(0, c1Value);
9         assertEquals(5, c2Value);
10
11         c1.increment();
12         c2.increment();
13         c2.increment();
14         assertEquals(1, c1.getValue());
15         assertEquals(7, c2.getValue());
16
17         c1.increment(3);
18         c2.increment(3);
19         assertEquals(4, c1.getValue());
20         assertEquals(10, c2.getValue());
21     }
22 }
```

2.2 What You Are Required to Do

1. Inspect the tester class (Section 2.1):

- 1.1 Identify missing classes and Create empty classes.

Principle 1: On the left-hand side of a variable assignment (=), if the type refers to the name of some non-existing class, then you must create that class.

For example, Line 4 and Line 5 of **TestCounter** suggest that a new class **Counter** is needed for the declaration of variables **c1** and **c2**'s types. You should then start by first creating a new, empty class accordingly:

```
public class Counter { }
```

The actual lab or lab test might require you to create multiple new classes, but the same principle applies.

- 1.2 Also, be sure to add a line, importing the new class, to the JUnit test, e.g.:

```
package junit_tests;

import model.Counter;

public class TestCounter {
    ...
}
```

1.3 Identify and add method declarations to “empty class(es)” just for compilations.

1.3.1 Identify constructors.

Principle 2: On the right-hand side of a variable assignment (=), if there is a **new** keyword, then the class name that follows indicates a call to a constructor of that class.

For example, Line 4 and Line 5 of **TestCounter** suggest two versions of constructor for the **Counter** class (i.e., the constructor is *overloaded*): one version that takes no parameters, and the other version that takes an integer parameter.

Consequently, we should add these two constructor declarations (with no implementations) to the **Counter** class:

```
public Counter () { }  
public Counter (int value) { }
```

1.3.2 Identify accessors.

Principle 3: If a method call appears on the right-hand side of a variable assignment (=), or as the input of a JUnit assertion such as `assertEquals(1, c1.getValue())`, then that method should be an accessor method.

For example, Lines 6, 7, 14, 15, 19, and 20 of **TestCounter** suggest that **getValue** is an accessor method with no input parameters.

Q1. Which class should **getValue** be added to?

Look at the **context objects** of the method calls: **c1** and **c2** are declared of type **Counter**, so the **getValue** method should be declared there.

Q2. What should be the return type of **getValue**?

Look at lines such as Line 6 and Line 7, indicating types of variables storing the return values. Consequently, we should add the following accessor method declaration (which only returns a **default value**) to the **Counter** class:

```
public int getValue() {  
    int result = 0; /* 0 is the default value of the return type int */  
    return result;  
}
```

1.3.3 Identify mutators.

Principle 4: If a method call appears alone as the entire line, then that method should be a mutator method.

For example, Lines 11 to 13 and 17 to 18 suggest that **increment** is a mutator method.

More specifically, the **increment** is *overloaded*: Lines 11 to 13 suggest one version of **increment** that takes no parameters, whereas Lines 17 to 18 suggest a second version that takes an integer parameter.

Q1. Which class should **increment** be added to?

Look at the context objects of the method calls: **c1** and **c2** are declared of type **Counter**, so the **getValue** method should be declared there.

Q2. What should be the return type of **increment**?

All mutator methods have the **void** return type.

Consequently, we should add these one accessor method declaration (with no implementations) to the **Counter** class:

```
public void increment() { }  
public void increment(int value) { }
```

1.3.4 Based on the identification of the constructors, accessors, and mutators as described above, we end up with:

```
public class Counter {
    public Counter () { }
    public Counter (int value) { }
    public int getValue() {
        int result = 0; /* 0 is the default value of the return type int */
        return result;
    }
    public void increment() { }
    public void increment(int value) { }
}
```

Principle 5: The above expanded `Counter` class and the given `CounterTester` class now **compile**. However, running the JUnit class `TestCounter` will result in a **red bar** (more precisely, all tests fail as no method has been properly implemented).

1.4 **Complete implementations of methods (for producing the expected output).**

Principle 6: Complete implementations of all methods, by observing method calls in the JUnit tester class (Section 2.1) and their expected values specified in the assertions.

Additional attributes (class-level variables) and helper methods are allowed if considered necessary.

Consequently, here is the final working version of the `Counter` class:

```
public class Counter {
    /* attributes */
    private int value;

    /* constructors */
    public Counter () {
        value = 0;
    }
    public Counter (int value) {
        this.value = value;
    }

    /* accessors */
    public int getValue() {
        return value;
    }

    /* mutators */
    public void increment() {
        this.value ++;
    }
    public void increment(int value) {
        this.value += value;
    }
}
```

Note. You can assume that the JUnit test class will be placed inside the `junit_tests` package, whereas any new classes identified should be added to the `model` package.

3 Source Code

You can find here the example covered in the notes for practice:

- Starter: https://www.eecs.yorku.ca/~jackie/teaching/lectures/2021/F/EECS2030/notes/EECS2030_F22_Inferring_Classes_from_JUnit.zip
- Solution: https://www.eecs.yorku.ca/~jackie/teaching/lectures/2021/F/EECS2030/notes/EECS2030_F22_Inferring_Classes_from_JUnit_Solution.zip