Week 9 - Monday

COMP 3100

Last time

- What did we talk about last time?
- Software quality assurance

Questions?

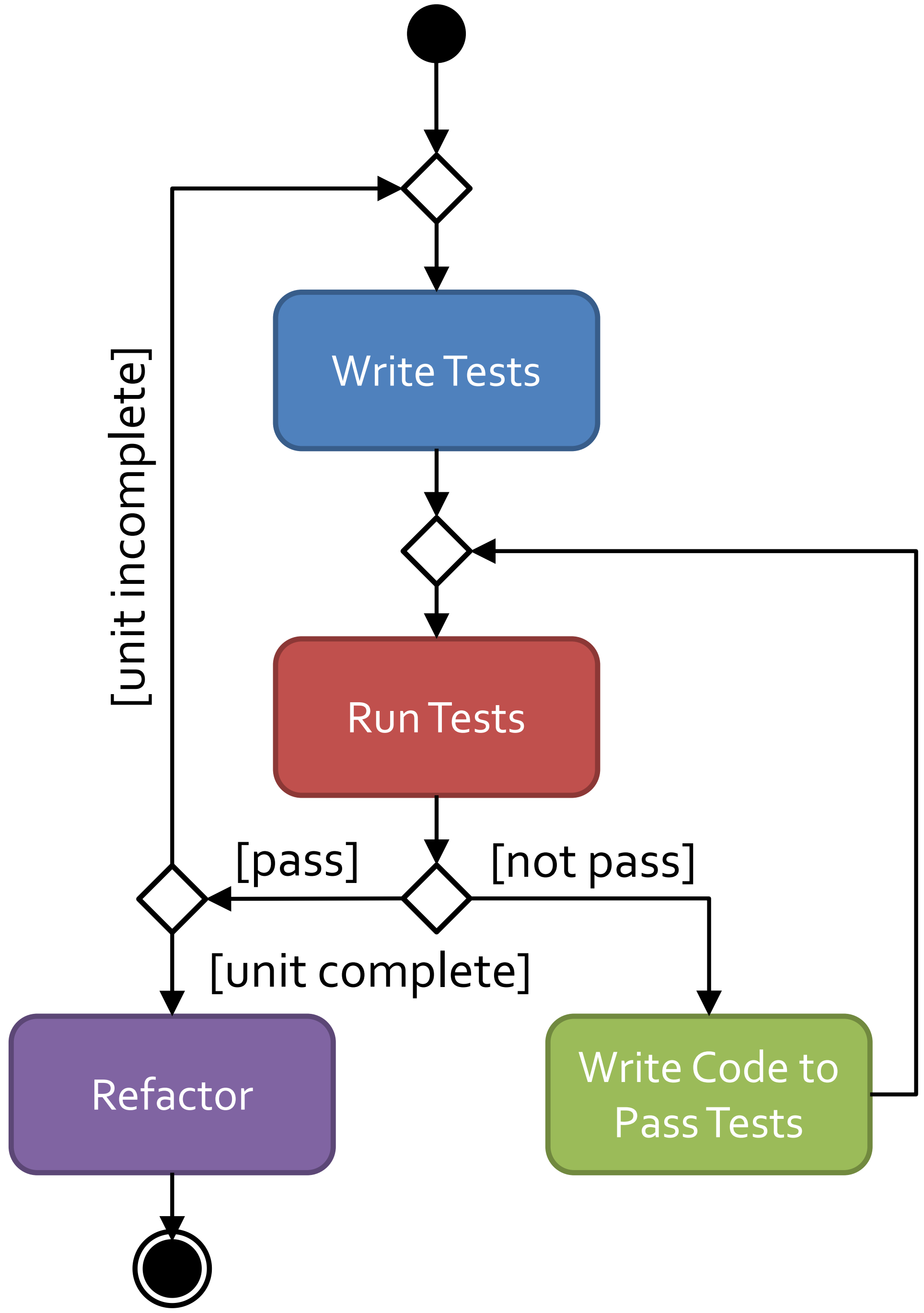
Test Driven Development

Test driven development

- **Test driven development** (TDD) is a style of development where testing is an integral part of coding
- The key idea of TDD is that you write tests for the code **before** you write the code
  - Thus, the tests aren't distorted by writing the code
- TDD is used for Extreme Programming, but it can be used for any approach, agile or plan-driven

Principles of TDD

- You have to have a testing framework
- Tests are written before code
- Tests and code are written incrementally
  - Write tests for some functionality, then write code to pass them
- Code is *only* written to pass tests
  - "Doing the simplest thing that could possibly work"
- Refactoring is expected
  - Writing code only to pass tests might end up with funky design



Benefits of TDD

- By making the test first, you really understand what you're trying to implement
- Your testing has better code coverage, testing every segment of code at least once
- Regression testing happens naturally
- Debugging should be easier since you know where the problem likely is (the new code added)
- The tests are a form of documentation, showing what the code should and shouldn't do

JUnit

Unit testing tools

- Nowadays, running large test suites can be automated
- Tools such as JUnit and other testing tools allow us to:
  - Write clearly marked tests with special set-up and clean-up code if needed
  - Run the tests, sometimes with randomized values or in randomized orders
  - Record which tests pass and fail
  - Show coverage information to see which lines of code the tests covered

JUnit

- JUnit is a popular framework for automating the unit testing of Java code
- JUnit is built into IntelliJ and many other IDEs
- It's possible to run JUnit from the command line after downloading appropriate libraries (but doing so is a pain)
- JUnit is one of many xUnit frameworks designed to automate unit testing for many languages
- You are required to make JUnit tests for Project 3
- JUnit 5 is the latest version of JUnit, and there are small differences from previous versions

JUnit classes

- For each set of tests, create a class
- Code that must be done ahead of every test has the **`@BeforeEach`** annotation
- Each method that does a test has the **`@Test`** annotation

```
import org.junit.jupiter.api.*;
public class Testing {

        private String creature;

        @BeforeEach
        public void setUp() {
                creature = "Wombat";
        }

        @Test
        public void testWombat() {
                Assertions.assertEquals("Wombat", creature, "Wombat failure");
        }
}
```

Assertions

- An assertion is something that **must** be true in a program
- Java (4 and higher) has assertions built in
- You can put the following in code somewhere:

```
String word = "phlegmatic";
assert word.length() < 5 : "Word is too long!";
```

- If the condition before the colon is true, everything is fine
- If the condition is false, an **`AssertionError`** will be thrown with the message after the colon
- Caveat: The JVM normally runs with assertions turned off, for performance reasons
- You have to run it with assertions on for assertion errors to happen
- You **should** run the JVM with assertions on for testing purposes

Assertions in JUnit tests

- When you run a test, you expect to get a certain output
- You should assert that this output is what it should be
- JUnit 5 has a class called **`Assertions`** that has a number of static methods used to assert that different things are what they should be
  - Running JUnit takes care of turning assertions on
- The most common is **`assertEquals()`**, which takes the expected value, the actual value, and a message to report if they aren't equal:
  - **`assertEquals(int expected, int actual, String message)`**
  - **`assertEquals(char expected, char actual, String message)`**
  - **`assertEquals(double expected, double actual, double delta, String message)`**
  - **`assertEquals(Object expected, Object actual, String message)`**
- Another useful method in **`Assertions`**:
  - **`assertTrue(boolean condition, String message)`**

Assertion example

- We know that the **`substring()`** method on **`String`** objects works, but what if we wanted to test it?

```
import org.junit.jupiter.api.*;

public class StringTest {

        @Test
        public void testSubstring() {
                String string = "dysfunctional";
                String substring = string.substring(3,6);
                Assertions.assertEquals("fun", substring, "Substring failure!");
        }
}
```

Sometimes failing is winning

- What if a method is **supposed** to throw an exception under certain conditions?
- It should be considered a failure **not** to throw an exception
- The `Assertions` class also has a `fail()` method that should never be called

```
import org.junit.jupiter.api.*;

public class FailTest {
      @Test
      public void testBadString() {
            String string = "armpit";
            try {
                  int number = Integer.parseInt(string);
                  Assertions.fail("An exception should have been thrown!");
            }
            catch(NumberFormatException e) {}
      }
}
```

JUnit practice

- Imagine you've got a method that decides whether a year is a leap year

```
public static boolean isLeapYear(int year)
```

- What are good tests for it?
- Let's write at least three JUnit tests for it
- We can do TDD and write the method *after* we write the tests

JUnit practice

- Imagine you've got a method with the following signature that sorts an array in ascending order

```
public static void sort(int[] array)
```

- What are good tests for it?
- Let's write at least four JUnit tests for it
- We can do TDD and write the method *after* we write the tests

JUnit practice

- Imagine you've got a class that stores time

```
public class Time {
        private int hour;
        private int minute;
        private boolean am;
        // Methods
        public Time(int hour, int minute, boolean am) {}
        public String toString() {} // Example: "3:06 pm"
        public int getHour() {}
        public int getMinute() {}
        public boolean isAm() {}
        public void addMinutes(int minutes) {}
        public void addHours(int hours) {}
}
```

- What are good tests for it?
- Let's write at least four JUnit tests for it

Upcoming

Next time…

- Debugging and system testing on Wednesday

Reminders

- Read Chapter 10: System Testing for Wednesday
- Finish the final version of Project 2 and the reflection
  - **Due tonight before midnight!**