Week 13 - Wednesday

COMP 2000

Last time

- What did we talk about last time?
- Software engineering
- Modeling and UML
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - State diagrams
  - Class diagrams
- Architecture patterns

Questions?

Project 4

Testing

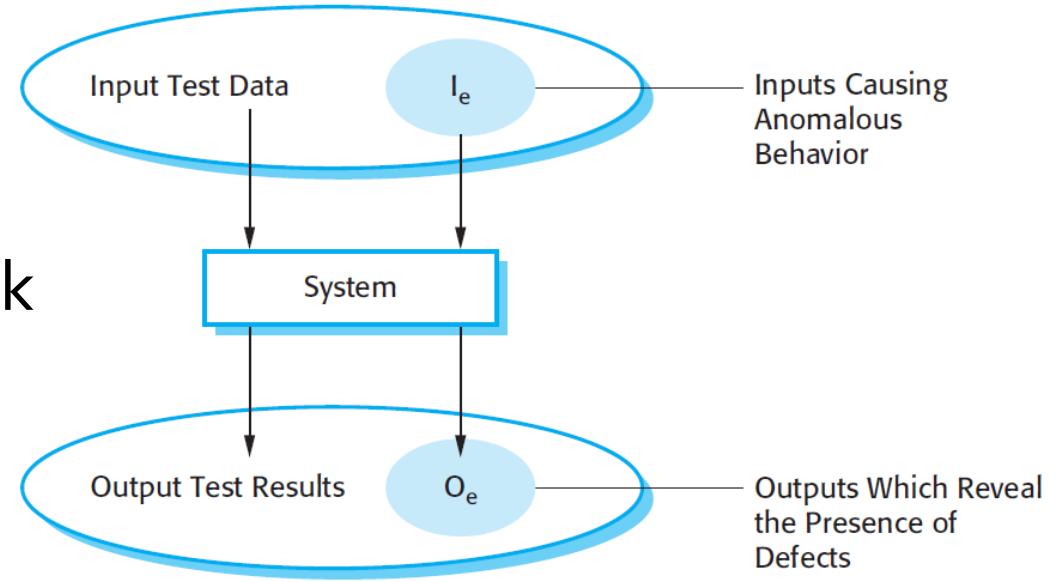
Testing

- *Testing can only show the presence of errors, not their absence.*
  - Edsger Dijkstra

- Historically, testing has sometimes been a task given to junior developers
- As systems have gotten more complex, people have gotten more excited about testing
- Finding ways to test subtle aspects of a program can be a rewarding challenge

Purposes of testing

- There are two almost opposing purposes for testing
- Showing that software meets its requirements
  - **Validation testing**
  - Looking for good outputs
- Finding inputs where software doesn't work
  - **Defect testing**
  - Looking for bad outputs
- When a project is due, students often confuse the two
  - Trying to convince themselves that the code is fine instead of looking for problems



Stages of testing

- Commercial software systems often go through three stages of testing
- Development testing
  - Look for bugs during development
  - Designers and programmers do the testing
- Release testing
  - Test a complete version of the code to see if it meets requirements
  - A separate testing team does the testing
- User testing
  - Users test the system in a real environment
  - Acceptance testing is a special kind of user testing to decide whether or not the product should be accepted or sent back

Development testing

- Development testing is the idea of testing you're most familiar with
  - Testing the software as it's being developed
  - Development testing is focused on defect testing
  - Debugging happens alongside development testing
- Three stages of development testing:
  - Unit testing: testing individual classes or methods
  - Component testing: testing components made from several objects
  - System testing: testing the system as a whole

Unit testing

- Unit testing focuses on very small components
  - Methods or functions
  - Objects
- Unit tests try many different inputs for the methods or objects to make sure that the outputs match

Unit test example

- Broken method to determine if a year is a leap year:

```
public static boolean isLeapYear(int year) {
    return year % 4 == 0 && year % 100 != 0;
}
```

- Tests:
  - `isLeapYear(2016)` → `true` (correct)
  - `isLeapYear(2018)` → `false` (correct)
  - `isLeapYear(1900)` → `false` (correct)
  - `isLeapYear(2000)` → `false` (incorrect)

Automated tests

- Because unit tests are based on simple relationships between input and expected output, they can usually be automated
  - **And they totally should be**
- Automated tests have three parts:
  - Setup: initialize the system with the inputs and expected outputs
  - Call: call the method you're testing
  - Assertion: compare the real output with the expected output

Automated testing frameworks

- You can create unit tests by hand and run them
- However, the problem is so universal that many automated testing frameworks have been created
- The most famous for Java is JUnit
  - Wikipedia lists about 50 just for Java
  - Some have special strengths, like creating mock objects that behave in ways that are useful for testing
- These testing frameworks make it easier to generate and run the tests

Choosing unit test cases

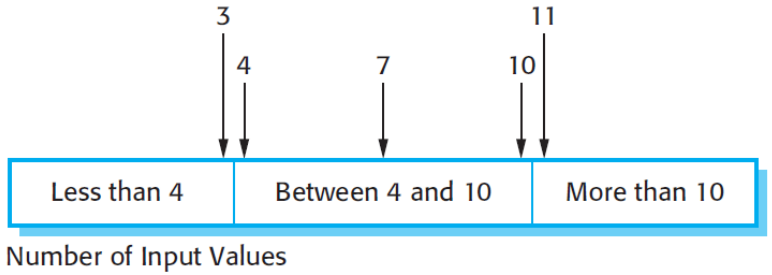
- Effective tests will show:
  - When used as expected, a component does what it's supposed to
  - Defects, if there are any, in a component
- In testing terminology, these are called **positive tests** (showing that stuff works) and **negative tests** (trying to make things crash)
- It's hard to pick good test cases

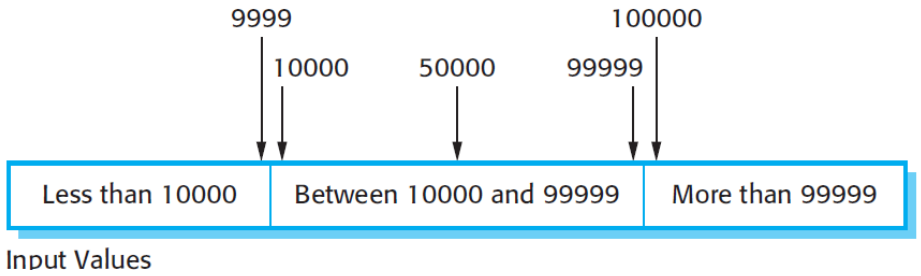
Choosing unit test cases, continued

- Two strategies for picking test cases:
  - Partition testing
    - Identify groups of inputs that will be processed in the same way
    - Pick representatives from each group
  - Guideline-based testing
    - Use guidelines to choose test cases
    - Guidelines are based on experience about the kinds of errors that programmers often make

Equivalence partitioning

- Partition testing is based on the observation that programs often behave similarly for all members of a set of values
- Such a set is called an **equivalence partition**
- You can try to find a set of equivalence partitions that covers all behaviors





Examples of guidelines

- When dealing with sequences, arrays, and lists, consider:
  - Testing software with sequences that have a single value (or no values)
  - Use sequences of different sizes in different tests
  - Design tests that access the first, middle, and last elements of a sequence

Black box testing

- One philosophy of testing is making **black box tests**
- A black box test takes some input **A** and knows that the output is supposed to be **B**
- It assumes nothing about the internals of the program, only the specification
- To write black box tests, you come up with a set of input you think covers lots of cases and you run it and see if it works
- In the real world, black box testing can easily be done by a team that did not work on the original development

White box testing

- **White box testing** is the opposite of black box testing
  - Sometimes white box testing is called "clear box testing"
- In white box testing, you can use your knowledge of how the code works to generate tests
- Are there lots of if statements?
  - Write tests that go through all possible branches
- There are white box testing tools that can help you generate tests to exercise all branches
- Which is better, white box or black box testing?

Component testing

- Beyond unit testing is **component testing**
- Components are made up of several independent units
- The errors are likely to be from interactions between the units
  - Hopefully, the individual units have already been unit tested
- The interfaces between the units have to be tested
  - Parameter interfaces in method calls
  - Shared memory interfaces
  - Procedural interfaces in which an object implements a set of procedures
  - Message passing interfaces

System testing

- System testing is when we integrate components together in a version of the whole system
- Though similar to component testing, there are differences:
  - Older reusable components and commercial components might be integrated with new components
  - Components developed by different teams might be integrated for the first time
- Sometimes, you only see certain behavior when you get everything together
- Try testing all the use cases you expect the system to see

JUnit

JUnit

- JUnit is a popular framework for automating the unit testing of Java code
- JUnit is built into Eclipse and many other IDEs
- It is possible to run JUnit from the command line after downloading appropriate libraries
- JUnit is one of many xUnit frameworks designed to automate unit testing for many languages
- You are required to make JUnit tests for Project 4
- 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 with the following signature that can determine whether or not a **`String`** is a palindrome
  - Assume it's a sophisticated function that ignores case and punctuation

```
public static boolean isPalindrome(String phrase)
```

- What are good tests for it?
- Let's write at least four JUnit tests for it, covering cases when **`phrase`** is a palindrome and when it isn't

Complex class

- Consider the following **`Complex`** class, for holding real and imaginary numbers

```
public class Complex {
        final double real;
        final double imaginary;
        public Complex(double real, double imaginary) {
                this.real = real;
                this.imaginary = imaginary;
        }
        public double getReal() {
                return real;
        }
        public double getImaginary() {
                return imaginary;
        }
}
```

JUnit practice

- Let's make a **`quadratic()`** method that returns an array of **`Complex`** objects that are the roots of the quadratic equation $ax^2 + bx + c$

```
public static Complex[] quadratic(double a, double b, double c) {
      // Fill in code
}
```

- Now, let's test it!
- What are good test cases?

Quiz

Upcoming

Next time…

- More JUnit examples

Reminders

- Work on Project 4