Test-Driven Development

CSE 219
Stony Brook University
Worker Hours

- Labor is often measured in worker-hours, worker-months, worker-years, etc.
- A famous example from the yesteryears of the gaming industry, “Doom”
  - Originally announced as Doom 4 in 2008
  - Finally released in 2016 simply as “Doom”
  - Took 8 years and more than 100 person-years of labor to develop
  - When asked in 2012, the company spokesperson and CEO John Carmack said that it will be “done when it’s done” … and the game was very positively upon its delayed release

But why not double the size of the team and halve the time?
The Mythical Man-Month

- If a software takes one expert programmer a year to develop
- And due to market pressures, the company wants the product in a month

\[
\text{1 developer x 12 months} =\text{12 developers x 1 month?}
\]

- Throwing additional developers into a project that is late is likely to delay a project even further. Instead,
  - Features that were promised but not yet completed
  - Don’t multiply worker bees, but make sure there is at least one queen bee … the worker who has detailed knowledge of all the modules
Design → Implementation

● When a modular design has been completed, ask these questions:
  ○ Can all the modules be developed in parallel?
    ■ Unlikely, because usually, there are dependencies
  ○ Can classes within a module be developed in parallel?
    ■ Unlikely, because usually, there are dependencies!
  ○ Can members (methods, etc.) within a class be developed in parallel?
    ■ Unlikely, because usually, there are dependencies!!

● Two approaches
  ○ Top-down
  ○ Bottom-up
Top-down Development

- Emphasize planning and a complete overall understanding of the system
- No coding begins unless and until adequate details have been reached in the design (at least some part of the system, if not entirely)
  - Implementation begins by attaching test stubs, which are programs that simulate the behavior of some software component
- Delays testing of functional units
- This was the traditional approach
  - Stopped being the most popular approach in the mid-1980s
Bottom-up Development

● Emphasizes coding and testing as soon as the first module is specified
  ○ May lead to a situation where a module is being developed without a very clear idea of how it links to other modules in the larger system
  ○ As a designer and developer, you are forced to think abstractly, and in general terms
  ○ This usually leads to reusable code, and the resulting system is highly decoupled!

● These days, we usually combine top-down and bottom-up approaches
  ○ A project being developed in isolation may be mostly top-down
  ○ Otherwise, developers will try to leverage pre-existing modules, giving the development a bottom-up flavor
Development Strategy

● Primarily top-down in a theoretical sense
● Bottom-up used when pre-existing modules can be reused or when developing a module is potentially simpler than developing a simulator

*Top-Down > Bottom-Up*

○ Simplifies system integration and testing
○ Critical high-level errors are easier to catch early on
What is Test-Driven Development?

- Design modular classes and methods
- Before coding, decide
  - What needs to be coded
  - What needs to be tested
  - Design test cases for important methods
- While coding
  - Test incrementally, as you implement your solution
What is Test-Driven Development?

```plaintext
assert !(test driven development) == doom;
```

- **Things not to do**
  - Coding without design
  - Designing without a plan for testing
  - Creating large amounts of untested code
    - Some experienced developers will tell you that 10 lines of untested code qualifies as “large amount”
  - Writing large methods, which typically indicates lack of modularity
    - Remember the Spaghetti and the Blob!
Testing and Debugging

● **Testing** is the process of running a program on a set of test cases and comparing the actual against the expected results.
  ○ **Testing** is meant to find defects in code and experimentally verify that the program does what it is supposed to do. It can be manual or automated.
  ○ There are different types of testing: unit, integration, system/acceptance, stress, load, etc.
  ○ **Debugging** is the process of finding and removing a specific bug from the program. It is always a manual, one-off process, since all bugs are different.

● **Defensive programming** is the philosophy of writing programs in a way such that debugging and testing is easier.
Levels of Testing

- **Unit testing** is the act of testing each module separately, in isolation
- **Integration testing** tests the interfaces between various modules
  - Much harder than unit testing
- **Regression testing** is done after modifications to ensure that the correct behavior of the original program is preserved
- **System testing** tests the overall behavior in an integrated environment
  - Evaluates the system’s compliance with specified requirements
How to Test? Black-box Testing

- It is impossible to perform exhaustive testing!
- Test cases are sampled
  - A small but representative sample of all input combinations
  - This is often done by what is called **black-box testing**, which is the process of testing using cases generated based on the program specifications, and not the implementation.
    - Tests are based on “what” the software is supposed to do but not “how” it does so
  - As a methodology, black-box testing can be applied to almost any type of testing, including unit, integration and system testing.
    - System testing is usually considered black-box testing since it should require no knowledge of the internal design to perform system testing
Black-Box Test Design Techniques

- **Use Case Testing**
  - where test cases are sampled in order to cover the system on a transaction by transaction basis from start to finish

- **Boundary-Value Analysis**
  - A boundary value is an input that is right at the edge of the set of standard acceptable input values for a system. Moving even the slightest bit over to the “other side” will produce different behavior from the system.
  - For example, how will a program that adds integers perform when the sum is exactly equal `Integer.MAX_VALUE`?
Glass-Box Testing*

- Tests the internal workings and structure of the system
  - Tests the “how” of the software, not the external behavior
  - Unit, integration, and system testing can be done with glass-box testing
  - + Due to internal knowledge of code, can zero in on weaknesses
  - + Makes tests traceable from source
  - + Are easy to automate, and hence much more testing can be done
  - - Can only be done by someone with internal knowledge of the code
  - - Focuses on existing functionality, so missing features may not be discovered

*Also called white-box, clear-box, transparent-box, or structural testing.
Unit Testing for Java Programs

- Test small units of a Java application
  - classes and methods
- Unit testing is usually an automatic process
  - Once implemented, can be run again and again
  - Make the machine work for you!
- There are several unit testing frameworks in Java
  - JUnit ← this is our weapon of choice
  - Arquillian
  - TestNG
A Simple Unit Test

- The class we want to test:

```java
public class StringUnit {
    public String concatenate(String a, String b) {
        return a + b;
    }
}
```

- To test a class, we must test all its public methods
  - There is only such method, `concatenate()`
  - So we will have a test method, `testConcatenate()`
    - Usually, a test method tests a single method in the target class
A Simple JUnit Test

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class StringUnitTest {

    @Test // tells the test runner that this method should be executed
    public void testConcatenate() {
        StringUnit myUnit = new StringUnit();
        String result = myUnit.concatenate("one", "two");
        assertEquals("onetwo", result); // if unequal, exception is thrown
    }
}
```
Assertion

- Assert methods form the secret ingredient for good unit testing in Java
- Important methods in `org.junit.Assert`
  - `assertArrayEquals()`
  - `assertEquals()`
  - `assertTrue()` and `assertFalse()`
  - `assertNull()` and `assertNotNull()`
  - `assertSame()` and `assertNotSame()`
  - `assertThat()`
Assertion

- Assert methods form the secret ingredient for good unit testing in Java
- Important methods in `org.junit.Assert`
  - `assertArrayEquals()`
    - are two arrays equal?
  - `assertEquals()`
    - are two objects equal, as per their `equals()` method?
  - `assertTrue()` and `assertFalse()`
    - is a variable’s value either true or false?
  - `assertNull()` and `assertNotNull()`
    - is a variable null or not null?
Assertion

- Assert methods form the secret ingredient for good unit testing in Java
- Important methods in `org.junit.Assert`
  - `assertSame()` and `assertNotSame()`
    - Are two object references pointing to the same object or not?
      - It is not enough that the two objects pointed to are equals according to their `equals()` methods. The exact same object must be pointed to.
  - `assertThat()`
    - Compares an object to an `org.hamcrest.Matcher` to see if the given object matches whatever the matcher requires it to match.
Matchers

● Matchers used to be an external addition to JUnit
  ○ JUnit 4.8.2 onwards, they are included by default

Matchers are used with the `assertThat()` method, which look like

```java
public void assertThat(Object o, Matcher matcher) { ... }
```

● There are several useful matchers readily available

● You can also make your own matchers
  ○ Unfortunately, Netbeans does not yet offer support for JUnit 5, which is fully compatible with
    unit testing for lambda expressions … where many of these external matchers are redundant.
**Matchers**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>matches anything</td>
<td>any0f()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>takes an array of matchers, and all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matchers must match the target object</td>
</tr>
<tr>
<td>is()</td>
<td>checks if objects are equal</td>
<td>instanceOf()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>checks if the given object is of a certain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type, or compatible with that type</td>
</tr>
<tr>
<td>describedAs()</td>
<td>adds a description to a matcher</td>
<td>sameInstance()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>checks whether the given objects is the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exact same instance as another</td>
</tr>
<tr>
<td>allOf()</td>
<td>takes an array of matchers, and all must match</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the target object</td>
<td></td>
</tr>
<tr>
<td>not()</td>
<td>negates the output of the previous matcher</td>
<td></td>
</tr>
</tbody>
</table>
Matcher Example

@Test
public void testWithMatchers() { assertThat(123, not(is(345))); }

public static Matcher matches(final Object expected) {
    return new BaseMatcher() {
        protected Object theExpectedObject = expected;

        public boolean matches(Object o) { return theExpectedObject.equals(o); }

        public void describeTo(Description description) {
            description.appendText(theExpectedObject.toString());
        }
    };
}

// usage: assertThat(myUnit.get(), matches("onetwo"));
IO Testing

- IO components are usually either read from an InputStream or written to an OutputStream (or some other similar components)
- In order to test these, you need control over the data that is read or written
  
  **IO Testing can be tricky!**

- Let us look at two tasks separately
  - Testing Input Components
  - Testing Output Components
public class TokenUnit {
    List<String> tokens = new ArrayList<>();

    void read(InputStream input) throws IOException {
        // method parameter: InputStream
        StringBuilder builder = new StringBuilder();

        int data = input.read();
        while (data != -1) {
            if (((char) data) != ',')
                builder.append((char) data);
            else {
                tokens.add(builder.toString());
                builder.delete(0, builder.length()); // empty the string builder
            }
            data = input.read();
        }
    }
}
import org.junit.Test;
import static org.junit.Assert.*;
import java.io.*;

public class TokenUnitTest {
    @Test
    public void testRead() throws IOException {
        TokenUnit unit = new TokenUnit();
        byte[] data = "123,456,789".getBytes();  // convert String to byte[]
        InputStream input = new ByteArrayInputStream(data);
        unit.read(input);
        assertEquals("123", unit.tokens.get(0));
        assertEquals("456", unit.tokens.get(1));
        assertEquals("789", unit.tokens.get(2));
    }
}
Testing Output Components

- A “unit” that writes to an output stream

```java
public class TokensOutputUnit {

    List<String> tokens = new ArrayList<String>();

    public void write(OutputStream output) throws IOException {
        for(int i = 0; i < tokens.size(); i++) {
            if(i > 0)
                output.write(',');
            output.write(tokens.get(i).getBytes());
        }
    }
}
```
public class TokensOutputUnitTest {

    @Test
    public void testWrite() throws IOException {
        TokensOutputUnit unit = new TokensOutputUnit();
        ByteArrayOutputStream output = new ByteArrayOutputStream();

        unit.tokens.add("one");
        unit.tokens.add("two");
        unit.tokens.add("three");
        unit.write(output);

        String string = new String(output.toByteArray()); // string created from byte array
        assertEquals("one,two,three", string);
    }
}
Running a Program with Assertions

- By default, assertions are disabled at run time
- Enabling them is done by a switch parameter
  -\(-\text{enableassertions} \mid -ea\)
- From the command line, run your test as
  \(\text{java } -ea \text{ TokensOutputUnitTest}\)
- Assertions can be selectively enabled/disabled at class and package levels
  \(\text{java } -ea: \text{packagename} -da: \text{SomeOtherClass TokensOutputUnitTest}\)

Switch to disable assertions is \(-\text{disableassertions} \mid -da\)
Exception Handling or Assertion?

- This is not a matter of “choice”

- **Assertions should never replace exception handling!**
  - Exception handling is meant to deal with unusual circumstances during execution.
  - In other words, proper exception handling ensures **robustness**
  - Assertions are meant to ensure/test **correctness** of a program
  - Valid arguments that may be passed to a method are within the method contract.
  - Never use assertions to check validity of arguments!

```java
public void setRadius(double d) {
    assert d >= 0; // wrong; this should be done with exception handling
    this.radius = d;
}
```
Testing for Exceptions

● Sometimes, we may want to test whether or not a code throws the correct exception when given an invalid input

● There are two ways of doing this
  ○ Use annotation attributes
  ○ Wrap the test code in a try-catch block
Using Annotation Attributes for Testing

- Add the “expected” attribute to the @Test annotation

```java
@Test(expected = IllegalArgumentException.class)
public void testForExceptions() {
    TokenUnit unit = new TokenUnit();
    unit.throwIllegalArgumentException();
}
```

The test fails if an `IllegalArgumentException` is NOT thrown

- Doesn’t work if you want to test for multiple exceptions
Wrapping Test Code in try-catch

```java
@Test
public void testForExceptions() {
    TokenUnit unit = new TokenUnit();

    try {
        unit.throwIllegalArgumentException();
        fail("expected IllegalArgumentException");
    } catch (IllegalArgumentException ignore) {
        // ignore the expected exception
    }
}
```