# CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 4.2: Automated Testing

## In this Session

- Testing Procedures
- Testing Frameworks
- Test Coverage

## Terminology

- Unit Testing
- Regression Testing
- Black Box
- White Box
- Test-Driven Development (TDD)

## Common Testing Procedures

- Unit Testing
  Check that each “unit” (class) in the project behaves correctly.
- Integration Testing
  Check that components work together and that the interfaces between components work as expected.
- System Testing
  Does the sytem as a whole work correctly?
- (User) Acceptance Testing
  Do users of the system agree that the system does what it is supposed to do?

## Test Frameworks

- Libraries to make writing and running tests easier
- *x*Unit is prevalent — JUnit, NUnit, CPPUnit, PyUnit, ...
  - We will be using JUnit 4 (`http://junit.org/junit4`)
- Depending on how your software is structured, unit testing frameworks can also be used for other types of automated testing.
  - So, "unit testing" is sometimes incorrectly used to mean "any automated test".

## Regression Testing

During development, testing can help answer (at least) two questions:

1. Does the new stuff work?
2. Have we broken things which used to work?

## Black Box Testing

- The method / class / program has inputs and outputs, but the implementation is inside a box which you can't see into.
    - Picture a box with wires coming out.
- You test according to the specification.
    - What it is supposed to do.
    - Means tests are less likely to be shaped by your assumptions about how the implementation works.

## White/Glass Box Testing

- Testing designed with knowledge of the internal implementation.
- Means tests can pay special attention to cases where the implementation is complicated.
- Code coverage can become important here[1].

[1] More on code coverage later this lecture

## Test-Driven Development (TDD)

Roughly: *Write the tests before the code.*

Use the tests to determine when code is “ready”.

Note:

- Not the same as blindly hacking at code until the tests pass
    - You still need to think
    - Writing the tests is meant to make you think about how to design the code
- If you find a bug which your tests do not detect, add a test which does.
- Assumes you (or other programmers) are involved in writing tests. TDD does not mean outsourcing your thinking.

## JUnit 4

Tests are described in classes — one test class per real class.

```
import org.junit.Test;

public class BobTest {
    @Test
    public void test1(){
        ...
    }
}
```

- `@Test` is an annotation[2] used to tell JUnit that this method is a test which should be run.
- Tests take no parameters and return void.
- Tests should be named for what they are checking.

[2] like `@Override`

## An Arbitrary Class

```java
@Test
public void test1() {
    Bob b1 = new Bob();
    Bob b2 = new Bob(5);
    int k = b1.getValue(); // should be zero
    int j = b2.getValue(); // should be 5
}

@Test
public void test2() {
    Bob b1 = new Bob();
    Bob b2 = new Bob(5);
    int r = b1.compareTo(b2); // should be -1
}
```

- Unless these throw exceptions, this won't tell us much.

## Did it Work?

```
import org.junit.Assert;

@Test
public void test1() {
    Bob b1 = new Bob();
    Bob b2 = new Bob(5);
    int k = b1.getValue(); // should be zero
    Assert.assertEquals(0, k);
    int j = b2.getValue(); // should be 5
    Assert.assertEquals(5, k);
}
```

- If the assertion is not true, the test will fail.
- Test methods should only test **one** thing.
  - not *two* like in this example

## Setup and Teardown

Each test method should be *independent* but you may want common things set up before each test (e.g. `b1` and `b2`).

```
public class BobTest {
  private Bob b1, b2;
  @Before
  public void setup() {
    b1 = new Bob();
    b2 = new Bob(5);
  }

  @After
  public void teardown() {
    b1 = null;
    b2 = null;
  }
...
```

`@Before` will be run before *every* test.

`@After` will be run after *every* test.

## Assert

Some `static` methods in `Assert`:

- `assertEquals`
- `assertArrayEquals`
- `assertFalse` / `assertTrue`
- `assertSame` / `assertNotSame`
- `fail`

## Different import

The methods from Assert are static.

To avoid writing Assert. on everything, we can do this:

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

Can now just use the method names.

## Checking for Exceptions

Checking that an exception was thrown:

```
@Test(expected = EOFException.class)
public void exceptionTest() {
    callThatShouldThrowEOFException();
}
```

- Don't catch the exception, the test runner does
- Tell the test runner which exception it should catch
  - Note that `.class` is required to indicate the exception class file

## JUnit 4 – Running Tests

On the command line:

```
java org.junit.runner.JUnitCore testclass1 ...
```

In an IDE:

Blackboard guide and next week's prac

Notes:

- Need both the junit and hamcrest libraries on your classpath[3]
- JUnit is distributed separately from the Java SDK

So, on Debian[4]:

```
   java -Xbootclasspath/p:/usr/share/java/junit4.jar:\
/usr/share/java/hamcrest-all-1.3.jar\
 org.junit.runner.JUnitCore
```

[3] your IDE may take care of this

[4] and hence almost certainly Ubuntu as well

## What to Test?

Tests take time:

- . . . to write
- . . . to thoughtfully consider what should be tested
  - *may require some creative "malice"*
- . . . to execute
  - electricity isn't free
  - test machines can be in demand
    - especially if they overlap with "work" machines

Tests should

- . . . cover the input possibilities / features
- . . . (ideally) help to identify problems
- . . . not be orders of magnitude bigger than needed

## Common Things to Check

- Boundary cases:
    - 0 for numerical values
    - Empty sets/lists/arrays/...
    - `null`
    - `NaN`?
- Negative numbers
- Very large values / lines
- Inputs which can't be divided cleanly
- Names of non-existent resources
- Extant but non-permitted resources
- Anything you think will throw an exception
- *Things that work*

## Coverage

So you've made some white box tests

- How much of the code is actually tested?
- Of all the different ways your program could run, how many are *covered* by your tests?
- Of course, we'd like to not do more testing than we need to.

We'll look at three levels of coverage:

- Statement coverage
- Branch coverage
- Path coverage

Note that, while coverage is an important idea, we do not explicitly test coverage in the assignments in this course. It may, however, come up in the *exam*.

## Statement Coverage

```
int x = getLargestPrime();
System.out.println(x);
```

This code can be tested by executing once[5].

- There is only one possible sequence apparent in the code.

Running this once would give us statement coverage[6].

- Each statement has been run.

[5]Assuming `getLargestPrime()` doesn't use any hidden state

[6]Since there are no branches/paths, this also gives both branch and path coverage

## Branch Coverage

```
int max(int a, int b) {
    if (a < b) {
        return b;
    } else {
        return a;
    }
}
```

To ensure this code works correctly, we would need to test the decision (branch) in both true and false cases. If we do this for both “sides” of all decisions, that would give us branch coverage. In this case, we might test (as an example):
`a = 1, b = 2` and `a = 2, b = 1`[7].

[7]Note that these inputs would also give statement coverage, but not path coverage

## Hidden Decisions

Not all branches explicitly say `if` or `switch`.

```
int max(int a, int b) {
    return (a < b) ? b : a;
}
```

OR

```
int result = f(x) && g(x);
```

## Path Coverage — Loops

```
for (init; test; iterate) {
    body
}
nextStatement;
```

- init; test (fails); nextStatement;
- init; test; body; iterate; test (fails); nextStatement;
- init; test; body; iterate; test; body; iterate; test (fails); nextStatement;

The difference between the last two could be the difference between a working iterate and a faulty one.

Path coverage requires inputs which cause every possible path to be followed at least once.

- Usually, going through the loop 0, 1, and 2 times is sufficient.