Week 4.2 — Automated testing

Terminology

- Unit testing
- Regression testing
- Black box
- White box
- Test-driven development

Automated testing

- Unit testing
  Check that each "unit" (class) in the project behaves correctly.
- Integration testing[1]
  Check that components connect together properly. System testing[1] / Functional testing[2]
  Does the sytem as a whole do what it is supposed to do.
  (User) Acceptance testing[1]
  Does the client agree that the system does what it is supposed to do?

[1] Oxford Dictionary of Computer Science
[2] Textbook

Test frameworks

Libraries to make writing and running tests easier.
*x*Unit is prevalent — Junit, CPPUnit, PyUnit, ...
We will be using JUnit4 (`http://junit.org/junit4`).[3]
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".

[3] But, version 5. . . ? Shhhh.

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.
Means tests are less likely to be shaped by your assumptions about how the implementation works.

写 test的人 可以和写程序的人不一样

Whitebox testing

Testing designed with knowledge of the internals.
Means tests can pay special attention to cases which the particular implementation is likely to make tricky.
Code coverage can become important here.

Coverage

- Line coverage — each line tested?
- Branch coverage — is each branch (part of if/switch) tested
- Path coverage — is each path through the code tested

While coverage is an important idea, we do not explicitly test coverage in the assignments in this course.

Test-driven development

Roughly: *Write the tests before the code.*
Use the tests to determine when code is "ready".
Note:

- Not the same as blindly hack at code until the tests pass. (You still need to think).
- 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.

Quality Assurance — it's not tennis

The QA step (and its associated staff), do not mean that you don't have to test.
If a failure makes it to QA, it means

- *You* failed
- NOT QA will debug it for me.

JUnit4

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

```
import org.junit.Test;

public class BobTest {

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

- `@Test` is an annotation[4] 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.

[4] like `@Override`

An arbitrary class

```
@Test
public void test1() {
    Bob bob1=new Bob();
    Bob bob2=new Bob(5);
    int k=bob1.getValue();// should be zero
    int j=bob2.getValue(); // should be 50
}

@Test
public void test2() {
    Bob bob1=new Bob();
    Bob bob2=new Bob(5);
    int r=bob1.compareTo(bob2); // 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 bob1=new Bob();
    Bob bob2=new Bob(5);
    int k=bob1.getValue(); // should be zero
    Assert.assertEquals(0, k);
    int j=bob2.getValue(); // should be 50
    Assert.assertEquals(50, k);
}
```

If the assertion is not true, the test will fail.

Setup and teardown

Each test method should be independent but you may want common things set up before each test. Eg: `bob1` and `bob2`

```
public class BobTest {
  private Bob bob1, bob2;
  @Before
  public void setup() {
    bob1=new Bob();
    bob2=new Bob(5);
  }

  @After
  public void teardown() {
    bob1=null;
    bob2=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

Since the methods on `Assert` are `static` and we don't want to keep writing `Assert.` on everything, we can do this:

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

We can now just use the method names;

Checking for exceptions

If you want to check for exceptions, the following works[5]:

```
try {
    callThatShouldThrow();
    fail();
} catch (EOFException ef) { // squashing expected
}
```

[5] There are other approaches, see the doco

JUnit4 - Running tests

On the command line:

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

In an IDE:

tutors

Notes:

- You will need both the junit and hamcrest libraries on your classpath[6]
- JUnit is distributed separately from the Java SDK.

So, on Debian[7]:

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

[6] Your IDE may take care of this.

[7] 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

Not a check list

- 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-existant resources
- Extant but non-permitted resources
- Anything you think will throw
- *Things that work*

Escript example "Tensor multiplication"

$$A * B$$

where A and B are (independently) 1D(vectors), 2D(matricies), 3D, 4D.
We know each case uses different code, so we need to test[8]:

$1D * 1D$	$2D * 1D$	$3D * 1D$	$4D * 1D$
$1D * 2D$	$2D * 2D$	$3D * 2D$	$4D * 2D$
$1D * 3D$	$2D * 3D$	$3D * 3D$	$4D * 3D$
$1D * 4D$	$2D * 4D$	$3D * 4D$	$4D * 4D$

So $\{1, 2, 3, 4\} \times \{1, 2, 3, 4\}$.
How many of each type of problem? (2,2,2,2), (3,3,3,3), ... Non-"square"?

[8] I know some of these combinations don't make sense mathematically, but they still need to be tested.

Escript example

But the software can also run with multi-threading (1 thread is not the same as multi-threading off) AND/OR multi-process. So we need to consider $\{0T, 1T, 2T, \ldots\}$ and $\{0P, 1P, 2P, \ldots\}$.

$$\{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \times \{0T, 1T, 2T, \ldots\} \times \{0P, 1P, 2P, \ldots\}$$

is a lot for *one* type of operation.
We need to trim some cases but which ones?