

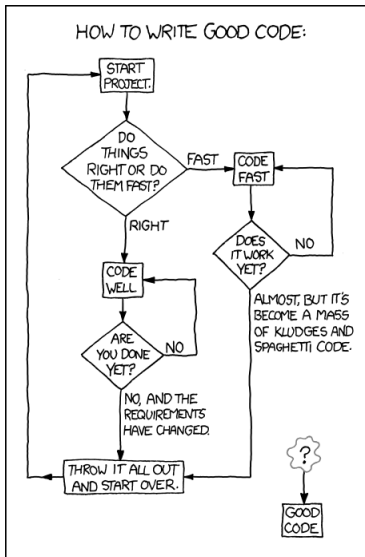
INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney



Lecture 16: Testing

*Types of testing, what needs testing, assertions,
unit tests*



Source: xkcd

It's not enough to be “confident” when you write your code that it just works: you have to *test* it.

Untested software is unreliable and often fails

Tested software can still be unreliable, but seldom fails

Thoroughly tested software is reliable and very rarely fails

Critical software systems such as (air-)traffic control, medical monitoring, construction, demand thorough testing. Testing is more significant than the code itself.

Documentation doesn't always help

```
1 return 1; # returns 1
```

or



```
/**  
 * Always returns true.  
 */  
public boolean isAvailable() {  
    return false;  
}
```

Never rely on a comment...

[link](#)

edited [Mar 16 at 18:11](#)

[community wiki](#)
[2 revs, 2 users 88%](#)
[martinus](#)

And just in case you needed more motivation

Microsoft employs at least as many testers as developers.

Just because it compiles...

“I’ve only got three compile errors: I’ll be finished soon.”

“It compiled with no errors! I’m done’

Compilation isn’t any kind of guarantee of correctness.

To test, we need need to inspect the contents of the memory

To test, we need see if the values of what we compute matches with what we expect

It is a very good exercise in working out, if something is wrong, what that problem is by *tracing through the code* itself: you can write out the state (value) of variables as you go through loops and if/elses ^[1]

This becomes time consuming with large amounts of code, however,

- the chance of identifying a problem is still very high
- the chance of making a mistake in code tracing (>1000 lines of code) also increases if not careful

^[1]a desk check!

What is a test?

If testing isn't just compiling, what *is* it?

A test gets some input to (part of) a program and then investigates the output.

Testing “HelloWorld.java” is easy:

input: `java HelloWorld` (or in fact, nothing)

output: “Hello, World!”

What is a test?

If testing isn't just compiling, what *is* it?

A test gets some input to (part of) a program and then investigates the output.

Testing “HelloWorld.java” is easy:

input: java HelloWorld (or in fact, nothing)

output: “Hello, World!”

Testing other things is more involved: e.g., testing a *squareRoot* method:

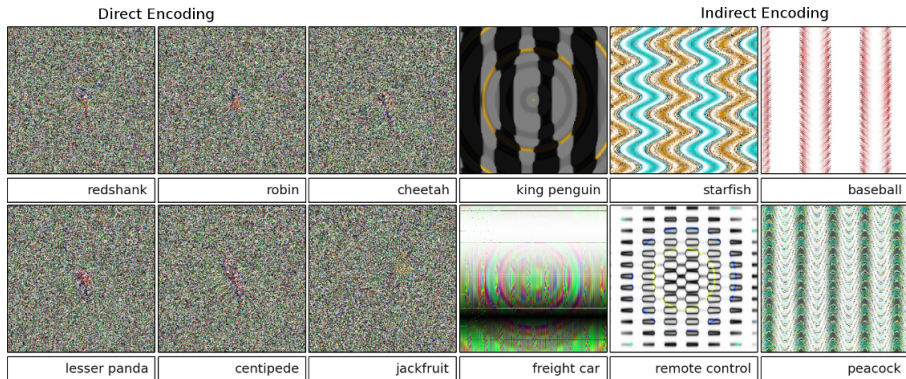
input: 4

output: 2

Now how do you test that the answer given (2) is the (positive) square root of 4? Let's look at testing some simple methods next.

An *oracle* is some external “thing” that we can ask for the *right* answer. We don’t always have this luxury.

When we do, we can check that our answer is the same as that of the oracle.



state-of-the-art DNNs trained on ImageNet believe with $\geq 99.6\%$ certainty to be a familiar object.

This is a way of software development that requires you to write the tests for a given method *first*, before you even write the method!

This has a number of advantages: it makes you think exactly what you want your method(s) to do in terms of

- 1 what the normal functionality should be
- 2 what should happen given “bad” input (e.g., passing a negative number to a method that expects positive numbers)

Failing a test is a *good thing*

It doesn't mean you've done a bad job coding (unless you never learn how to fix the code!)

- ❶ It means you've written a test
- ❷ It means you've found an error
- ❸ It even helps you *fix* the error

The test has brought your attention to a specific case where things should work.

The time spent on making the test, for a small part of the program, is much less than bug hunting the entire program.

Different flavours of testing

There are several approaches to testing.

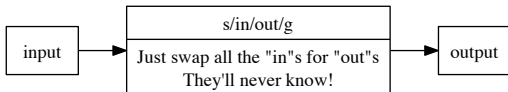
One is to treat the program as a kind of “black box” which simply processes input and gives output, with no reference to how it works... This “black box testing” is often quick and powerful.



Different flavours of testing

There are several approaches to testing.

One is to treat the program as a kind of “black box” which simply processes input and gives output, with no reference to how it works... This “black box testing” is often quick and powerful.



Alternatively we can look inside the box to see what's going on. This is known as “white box testing”

With black box testing you don't look inside each method: you just look at the input and output.

With white box testing you look inside the method in detail. This enables you to test different kinds of input and therefore different *execution paths*.

This is a very common method and also has its (limited) use:

Print out a running log of what's going on, while you're developing code, and print out error messages if things go wrong.

This is fine up to a point but *don't do it all the time!*

After finding the problems, you have to go back and comment out all those print statements, essentially all your checking/testing is never used again.

It's *far, far* better to have your testing separate from the rest of the code!

Tests are better if they're automated

I once read a developer (of some commercial software) state that the way he tests his code is to write it and then try as hard as he could to “break” it, **manually**.

His view was that as he should know the code best, he'd also know what were the most likely problems.

In general tests should be **automated**. This is so they can be *repeated*, because if it can't be repeated easily, how can you tell that a problem has been fixed?

Every time you fix something, you should re-run all your *previous* tests.

Suppose you have some code that calculates, say, the determinant of a matrix.

You have a test that works for a very simple matrix, which is 2×2 , like this:

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant of this matrix is given by $\det(M) = ad - bc$.

When you expand your code to cope with bigger matrices, you should *definitely* test that finding the determinant of the simple 2×2 matrix still works!

Regression testing (cont.)

Regression Testing is running ALL your tests again whenever you make a significant change to your code, to make sure you haven't broken code that used to work.

What if there are some *weird* inputs?

For example, what should happen when

- an empty list is given to a sorting method
- a negative number is given to a log method?
- a null is given to a print method?
- an index of -1 is given as the index of an item in an array?

Your code should cope sensibly with such things, and your tests should test that it does!

In engineering terms, an *edge* case is where one dimension of a problem is at an extreme, for instance when the load on a bridge is maximum.

A *corner case* is where more than one dimension is at an extreme, such as when a bridge has maximum load *and* it's being hit with highest possible winds *and* it's in an earthquake.

Design your code to cope with corners and edges! ☺

It should be obvious that you should test every possible part of your code, but how will you do that?

If you have a test for every *line* of your code, then you have great *coverage* (coverage is just the proportion of the code that is tested) — but that doesn't necessarily test every way in which your code will work.

Code coverage in testing is the proportion of (lines of) code that you have tested.

You need to think about each possible *execution path*

An execution path is a route through the flow of the program.

For example, if you have a loop, the execution path is either to go into the loop, or to iterate in it.

If you have an “if (a == 1)” statement, one execution path happens if a is 1, and another execution path happens if a is not 1.

An *assertion* is a condition (Boolean expression) written at some point in a program

We assert that the condition must be true whenever the program reaches that point

```
1  if( a > b )
2      m = a;
3  else
4      m = b;
5  //m = max(a,b)
```

There are no assertion keywords in Java, but there is a better way to handle this, Exceptions.

Precondition and postcondition

Preconditions and postconditions are *assertions* placed before and after some piece of code

```
1 // Pre : x >= 0
2 y = Math.sqrt(x);
3 // Post : y >= 0
```

The precondition asserts that x is non-negative at that point, which is needed to ensure that the `sqrt` function will work properly;

The postcondition asserts that y must equal \sqrt{x} afterwards, which is also true since otherwise something must be wrong with the *sqrt* function.

These communicate the idea that: this code will *always* produce a true postcondition if the precondition is true

An assertion of the class

Matryoshka doll: `size >= 0`

Can test a class invariant anywhere^[2] in the class and it is assumed to be true.

Most useful for identifying where a problem is, both from external or internal modifications of the object data

Not as helpful if instance variables are public: values of the object can change without using methods, from anywhere in the program at anytime

^[2]except in constructor before values are initialised

Testing a returned String

In many cases, e.g., “Hello World”, you return a `String` object. It’s easy to test whether a method returns the right string: we can just use compare the output string with what expect:

```
1  public String greet() {  
2      return "Hello, World!";  
3  }  
4  
5  public boolean isGreetOK() {  
6      if (greet().equals("Hello, World!"))  
7          return true;  
8      return false;  
9  }
```

which is a little clunky but does the job.

Here we are testing to see whether the answer is correct and returning `true` if (and only if) it is.

Testing a returned String (cont.)

Another way is to throw an error if the test is not passed.

```
1 public void testGreet() {  
2     if (!greet().equals("Hello, World!")) {  
3         throw new AssertionError("Greet is not working!");  
4     }
```

An Error is like an Exception in that it is throwable and it is handled and dealt with in a similar way

```
1 public static void main(String [] args) {  
2     try {  
3         throw new AssertionError("");  
4     } catch (Error e) {  
5         e.printStackTrace();  
6     }  
7 }
```

Testing a method that returns an int

Here's another simple method, that will return the area of a square with integer-valued sides, and a class with a method to test it:

```
1 public class Square {  
2     // ... lots of code in here to construct the Square instances etc.  
3     public int getArea() {  
4         return size * size;  
5     }  
6 }
```

```
1 import geometry.Square;  
2  
3 public class SquareTester {  
4     public boolean isAreaOK() {  
5         Square s = new Square(3); // assume constructor sets size  
6         if (9 != s.getArea()) {  
7             return false;  
8         }  
9         return true;  
10    }  
11 }
```

Testing a method that returns a floating point

Take care with methods that return floating points numbers. Should not compare floating point numbers with ==:

```
1 public class Circle {
2     // ...
3     public double getArea() {
4         return Math.PI* radius * radius;
5     }
```

This time I'll throw an exception if the value isn't correct:

```
1 public class CircleTester {
2     public void testCircleArea1() {
3         Circle c = new Circle(1.0); // sets the radius to 1.0
4         if (c.getArea() != 3.1415926) {
5             throw new AssertionError("The area is not correct"
6                                     + " for a circle of radius 1");
7         }
8     }
9 }
```

Testing a method that returns a floating point (cont.)

running this code fails even though my method is correct

rounding error is affecting the comparison

Instead of using `==` directly we test that the *difference* between the expected answer and the returned value is *small*:

```
1 public void testCircleArea2() {  
2     Circle c = new Circle(1.0); // sets the radius to 1.0  
3     if (Math.abs(c.getArea() - 3.1415926) > 0.000001) {  
4         throw new AssertionError("The area is not even"  
5             + " close for a circle of radius 1");  
6     }  
7 }
```

This code will only make the `Circle.getArea()` method fail if the area is more than 0.000001 different from what I expect.

avoid arbitrary numbers: `Math.ulp(x)` - Java machine floating point number epsilon

// Pre : $x \geq 0$

`y = Math.sqrt(x);`

// Post : $|y - \sqrt{x}| \leq 10^{-6}$