

Chapter 12

Testing

Testing

Testing should be done by the developers, it improves their performance.

Testing

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems

IEEE Software Engineering Body of Knowledge
www.swebok.org

Software testing consists of the **dynamic** verification of the behaviour of a program on a **finite** set of test cases, suitably **selected** from the usually infinite execution domains, against the **expected** behaviour.

Dynamic opposed to walk throughs, inspections etc.

Selected: key challenge is to identify tests which are most likely to expose failures. Multiple tests which challenge the same code are a waste of time to write and a waste of time to run.

Failure is undesired behaviour,
Fault is the cause of the failure

Scale: Unit; Component; Integration; System
Characteristic: Functional; Robustness;
Performance; Usability

Why?

Testing should be done by an independent team who have no preconceptions about its operation.

What?

Different system modelling and testing strategies are recommended for developing programmes in different contexts

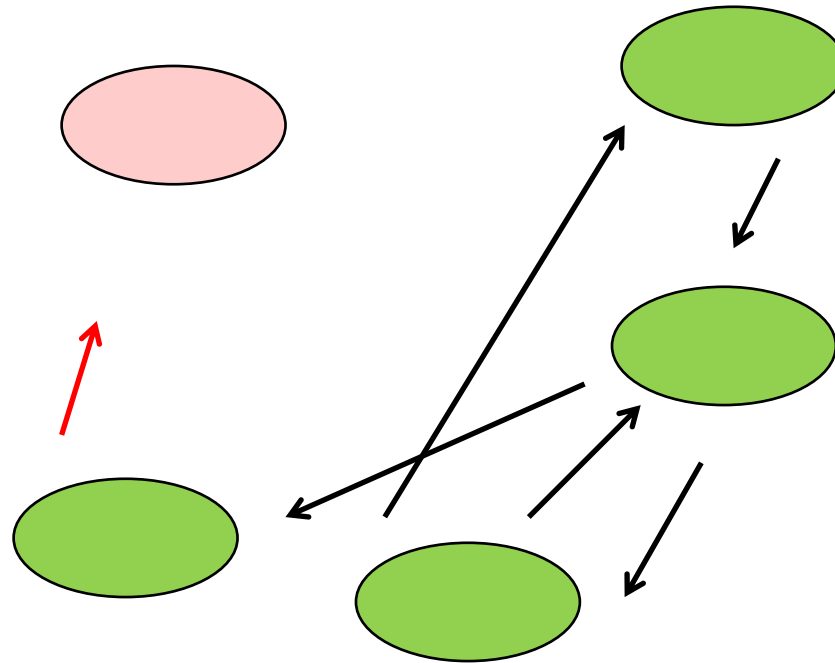
eg normal techniques are not suitable for real time programming where a timing is part of the specs

GC computing is distributed/parallel computing so we can apply the rules and techniques developed for those contexts.

GC is a new way of organising computing & represents a new set of problems which *may not have well developed solutions*.

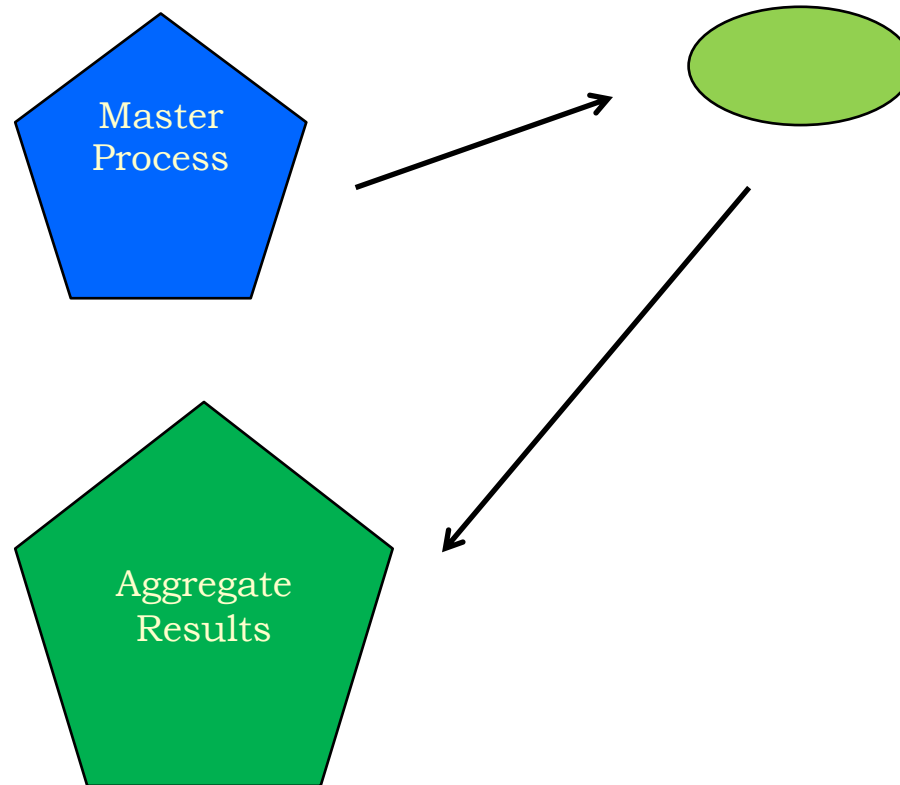
Obvious, but ...

Interacting systems with partial failure



What now?

Parallel system with task distribution



If the subtask does not return – then resubmit.
But what happens if after the duplicate task
completes the original task responds.

It is vital that the standard non-distributed testing procedures are carefully followed.

Conventional testing

White Box	Code logic visible to tester
Black Box	Provide input and verifies output

White Box	Unit tests (Integration tests)
-----------	-----------------------------------

Black Box	Unit tests Integration tests Functional tests Stress/Load tests Acceptance tests
-----------	--

Unit tests	Does the smallest operational unit work correctly (method)
Integration tests	Do the programming units (classes) work together
Functional tests	Do the functions/services provided by the application work in isolation

Stress/Load test	Operation in practice
Acceptance test	Is the sponsor/purchaser happy

People are much happier shipping untested code, then shipping code which fails

Test Driven Design (TDD)	
A technique which was pioneered by Kent Beck One of the founders of pattern based programming Track record	
Design the tests and then write the code to test Normal write the code and then write the tests. pressure to deliver code not to test it. pressure point is when code is required and the ability to say it is correct is at its worst	
Write the test(s) at the start when there is still lots of time. Testing becomes part of the development process, not an optional add-in. pressure point and the requirement is for the code to pass all the tests.	
TDD improves the code which is written <i>Not just because code has been tested.</i>	

Suitable for black box tests

Tests are automatic and self documenting.

TDD promotes
good coding
practice

Testing is part of the
development process not an
add-on

Test Driven Design (TDD)

Mistakes are spotted and corrected more quickly.
No time wasted moving down erroneous paths. Easier
to find error in 10 lines of code than 100

Rapid feedback between mistake and realisation of
mistake.

close loop means the solution is more closely
connected to the problem. Teachers told the
importance of rapid feedback and correction.

TDD encourages the creation of code that can be
tested.

If you write the tests then the code must pass the
tests.

Methods are likely to do one thing. The thing that you
test. Methods that do two or more things are harder to
test. **Methods should do one thing.**

Classes made up of focussed methods are more likely
to be focussed components. They concern one thing.
Classes which are testable are good object oriented
classes

Starting of course with unit
tests.

Complex number class

We need to write a complex number class.

Stage 1: Decide on the methods

- Constructor
- Add
- Multiply
- Modulus
- Complex Conjugate
- Equals

Stage 2: Decide on the tests

Does the constructor create the correct object?

Add Add two numbers with positive parts
 Add two numbers (+,+) and (+,-)

....

Add two equal and opposite numbers

equals

- Two equal numbers
- Two unequal numbers

Complex number class

Stage 3: Write the tests

Stage 4: Write enough of the class to run the tests

Stage 5: Run the tests ... all will fail.

Stage 6: write code until all tests pass

Most IDEs will generate tests if you give them the class with all its method signatures

```
Class CN {  
    CN() {  
    }  
    CN add(CN a) {  
        CN sum = new CN();  
        return sum  
    }  
    CN ComplexConjugate() {  
        CN cc = new CN();  
        return cc;  
    }  
}
```

Netbeans – given this will produce

```
public void testadd() {  
    CN A = new CN();  
    CN B = new CN();  
    CN C = A.add(B);  
}  
public void testCC() {  
    CN A = new CN();  
    CN B = A.CC();  
}
```

To get this to work we have to add the appropriate methods to the CN class.

Run will give errors

Need to add code to the tests

```
public void testCC() {  
    CN A = new CN(2.0,-3.0);  
    CN B = A.CC();  
    CN C = new CN(2.0, 3.0);  
    assertEquals(C.equals(B),true);  
}  
public void testsum() {  
    int Sum;  
    Sum = 1 + 1;  
    assertEquals(Sum, 2);  
}  
OR      assertEquals(Sum, 3);
```

Temptation – need to get state of numbers to check equality.

But we are doing Unit tests – not white box tests.

We do not need to see inside, nor should we attempt to.

We need to write an equals method and we can then use that..

How do we show equality – write the equals method of the complexNumber class

```
this.x = compare.x;  
this.y = compare.y;
```

Before we look at if add works check that equals work.

Typical of TDD.

Try to run a test – need some code.

That code needs testing.

Code passes test - return to original problem.

At the start you can find your self apparently moving back – but you are creating solid code and solid tests.

Test equality

Test inequality

Test equality to self

Test

Can feel frustrating – trying to test adhoc also ends up being frustrating

Implementing equals is often the first step in TDD (at least in the sort of problems I solve.

You find that you start multiplying tests.

They are easy to write and easy to run.

The right script (or even better the green bar if you run JUNIT standalone.

A pat on the back everytime it runs.

I can check just another possible problem place.

Only one equal

One set to zero

Both set to zero

.....

This is said to be being *test infected*

Considered to be a good thing

The process of writing and tested are integrated into one on going activity.

Now we can write tests for addition

Starting

Design a complex number.

What is the first test?

Create a complex number – is it correct?

How? Print out – but this is not a OO solution and will not work for repetitive tests?

Complex number what will it look like and can we test the simplest implementation

```
public complexNumber(){  
}  
public complexNumbe(double real, double imaginary){  
}  
public String toString(){  
}  
public boolean equals(Object comp){  
}  
public complexNumber clone() {  
}
```

toString returns a String so we can test that.

Actually we need to hack a few more lines to get it to compile

Note the equals
takes an object if
you want to
override it!

When overriding (overloading) equals remember it is an equivalence relation

- It is **reflexive**: for any reference value x, x.equals(x) should return true.
- It is **symmetric**: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is **transitive**: for any reference values x, y, and z, if
x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is **consistent**: for any reference values x and y, x.equals(y) always returns true or false. X,Y unchanged.
- For any non-null reference value x, x.equals(null) should return false.

Overriding **equals()** means you should override **hash()**
else users of hashtables etc. will get unexpected answers.

Else you will confuse
people (including yourself)

■ hash() returns an integer. Must be consistent during an invocation of an application – may vary across invocations.

■ Two objects for which equals(Object) is true must return the same hash code

■ It is **not required** that two objects for which equals(Object) is false, must return different hash codes. Better if they do.

Hash collisions.

Build hash from the state variables – creating good hash is hard.

If the state variables are objects a new hash can be created from the individual hash codes.

equals

overrides

```
public boolean equals(Object comp){  
}
```

overloads

```
public boolean equals(complexNumber comp){  
}
```

Note if you overload - problems if someone passes an object.

```
complexNumber cn1 = new complexNumber();  
complexNumber cn2 = new complexNumber();  
Object ob = new Object();
```

.....

```
cn1.equals(cn2);
```

does what you expect

```
cn1.equals(ob);
```

probably doesn't

Need to check that an object is of the correct class before you can make the comparison

over

```
overrides need to check the class type
public boolean equals(Object comp){
    if(comp.getClass() == this.getClass()) {
        go ahead
    } else {
        result = false;
    }
}
.....
```

But if the Object pointer comp is null the attempt to get the class will fail.

So we need to add a clause

```
if(comp != null) {
    go ahead
} else {
    result = false;
}
.....
```

....

Silly to check the class if we provide an overloaded method which is only called with a pointer of the `complexNumber` type.

So

```
boolean result = false;
if ((this.real == comp.real)&&(this.imaginary ==
                                comp.imaginary)) {
    result = true;
} else {
    result = false;
}
.....
```

Allows us to check an overloaded equals function ... and the do something in the overridden function becomes

```
complexNumber cn = (complexNumber)comp;
result = this.equals(cn);
```

So we only write the code once.

Remember we have to check for null in the overloaded function for the case of a direct call with a `complexNumber` pointer to a null object.

toString

Test

```
complexNumber A = new complexNumber();  
assert (A.toString().equals("0.0 + i 0.0"));
```

Not revealing state - does not say how we are storing it.

Ought to allow print out in polars – how?

Only one toString! – like a calculator **mode**

Should mode correspond to an object or the class. CLASS

```
private static int modeType;  
public static boolean setMode(String mode){  
}
```

What if we some sends a mode which is neither polar nor cartesian?

Throw an exception!

complexNumberUnrecognisedMode

Exception

```
public class complexNumberUnrecognisedMode extends
    Exception{
    public complexNumberUnrecognisedMode(){
        super();
    }
    public complexNumberUnrecognisedMode(String mess){
        super(mess);
    }
}
```

Ended up writing a test and an exception class before the actual class!

Now we have another test ...

```
complexNumber A = new complexNumber();
setMode("cartesian");
assert (A.toString().equals("0.0 + i 0.0"));
setMode("polar");
assert (A.toString().equals("0.0 * exp(0.0)");
```

This zero almost certainly involves a special case – we are going to need a standard case

```
...    complexNumber B = new complexNumber(1.0,1.0);  
        B.setMode("cartesian");  
        assert (B.toString().equals("1.0 + i 1.0"));  
        complexNumber.setMode("polar");  
        assert (B.toString().equals("1.4142 * exp(0.778)"));  
        complexNumber.setMode("polr");
```

how to check for an exception
sucessfully thrown.

**Do we have to check that the exception class works
as expected?**

Netbeans will actually create the exception class which I
outlined with no further input.

Building the foundations takes time –.

Show the tests running at this point

We haven't tested the parts but we have tested that the print
prints what we might expect from the constructor

Add

```
public void add(){  
}
```

How is this to be tested?

No peeking at the internals – so we need to check equality

```
complexNumber A = new complexNumber(3.6,2.8);  
complexNumber B = new complexNumber(-5.4,0.5);
```

Is this `A.add(B)` and A is now `A = A + B`

Or is it `C = A.add(B)`;

A is left unchanged and a new value is returned?

The idea that a variable is not altered but is created and then is unchangeable is easy to organise in Java

use `final`

is a commonly used paradigm in concurrent programming, if there is no possibility of changing the value of a variable it does not have to be protected by synchronise.

Thinking about the test makes use think again about what we are going to implement

Testing add

Assume return a new value.

```
complexNumber C = A.add(B);
```

```
complexNumber D = new complexNumber(-1.8, 3.3);
```

```
assert (C.equals(D));
```

What else might we test for the addition?

<div data-bbox="19 14 444 714"> <h1>Responsibility</h1> <p>If you pass something which is not a complexNumber the return is not defined. Could set false for instance Could throw an exception Could throw a run time error</p> </div> <div data-bbox="19 714 444 1413"> <p>User doesn't need to test</p> </div>	<div data-bbox="444 14 1506 399"> <p>For equals method we check that what is passed is suitable. Not null, a complexNumber</p> <p>Person calling it may well decide to check it is not null and it has the correct type. Can lead to multiple checking – a waste of time.</p> </div> <div data-bbox="444 399 1506 928"> <p>Should be written into the “contract” If you pass an object pointer to a complexNumber</p> <ul style="list-style-type: none"> •Return true if real & imaginary parts are equal •Return false if either is not equal <p>If you pass an object pointer</p> <ul style="list-style-type: none"> •Return true if a complexNumber and real and imaginary parts are equal •Return false if either part is not equal •Throw NullPointerException if pointer is null •Throw InvalidObjectType if not complexNumber </div>	<div data-bbox="1506 14 1926 1413"> <h1>Testing</h1> </div>
---	---	---

There are a number of extra problems which may arise in distributed processing.

Co-ordination between the parts must be explicitly built in.

In single processor (thread) programming calculations will occur (or will appear to occur) in a predefined order.

An expression which comes early in the programme will be evaluated before ones in a later expression.

There will be no way to guarantee order in separate sub-programmes, except by specifically synchronising them.

Can we define a “universal time” to synchronise the processes to?

The processor may actually reorder the operations and perform some in parallel, but the results will be delivered “as if” serial.

Non return

A crash in single machine will fail to deliver the result.
A crash in one of xxxx machines will fail to deliver a small fraction of the result – which may or may not be important.

A set of machines testing from primality.

Machine which finds factors fails to return result.

Incorrect

Machine finding shortest path for the travelling salesman problem fails to return.

Second shortest path found – probably acceptable.

So influence of non return depends crucially on context.

No general statement possible.

Solution to non-return. **Resubmit.**

Non return is indistinguishable from slow return.

May get multiple returns to the same sub-problem.

For problems above irrelevant.

For searching a database for matching to criteria *can* be crucial,

UDP v TCP

Empty return is clearly
OK

Affects at two levels.

In job (finite element analysis) – where the calculations of one process must be available to other processes.

Intra-job – where results must be combined at the conclusion of a set of jobs.

These complexities need:

- to be considered;
- suitable solutions
- Corrected operation demonstrated.

Normal testing must be followed rigorously.

See chapter on
workflow

Testing is important for correct operation of **all** code.

But you can often get away with debugging

It is vital for distributed code.

It **will** break if it is not tested.

Write and run is not an option