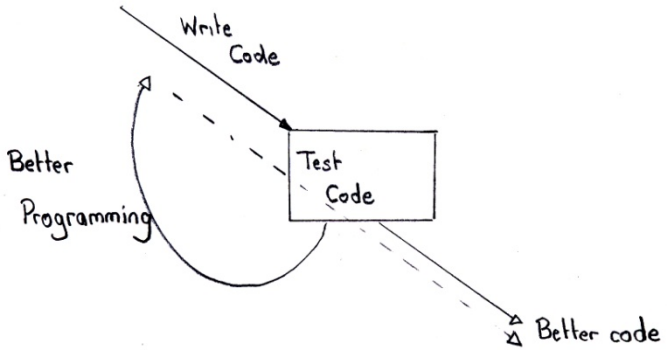
Chapter 6
Testing Quality

Improve Your Code

Correct code is produced by skillfull programmers

Reflective use of tests can improve coding quality



But only if the tests themselves are good.

How to measure and improve test quality?

Fulfills quality criteria

Measure tests

How much of the code is tested Coverage

The higher the coverage the better the test

all else being equal

Improving tests

If an error is reported by a user, then the test system has failed.

Having identified the source of the error a test should be written to isolate the error, so that it cannot recur.

If an integration test fails and the error is traced by to incorrect operation of an object or method, then a unit test should be written to detect this error.

Any error which is detected later than it could be, should generate a test at the appropriate level and errors should be detected as soon as possible

Coverage

Code which is not accessed is not tested.

Tools exist which measure which lines are executed by a test run.

Cobertura

Visual Studio (enterprise edition)

....

Code which is accessed may be tested.

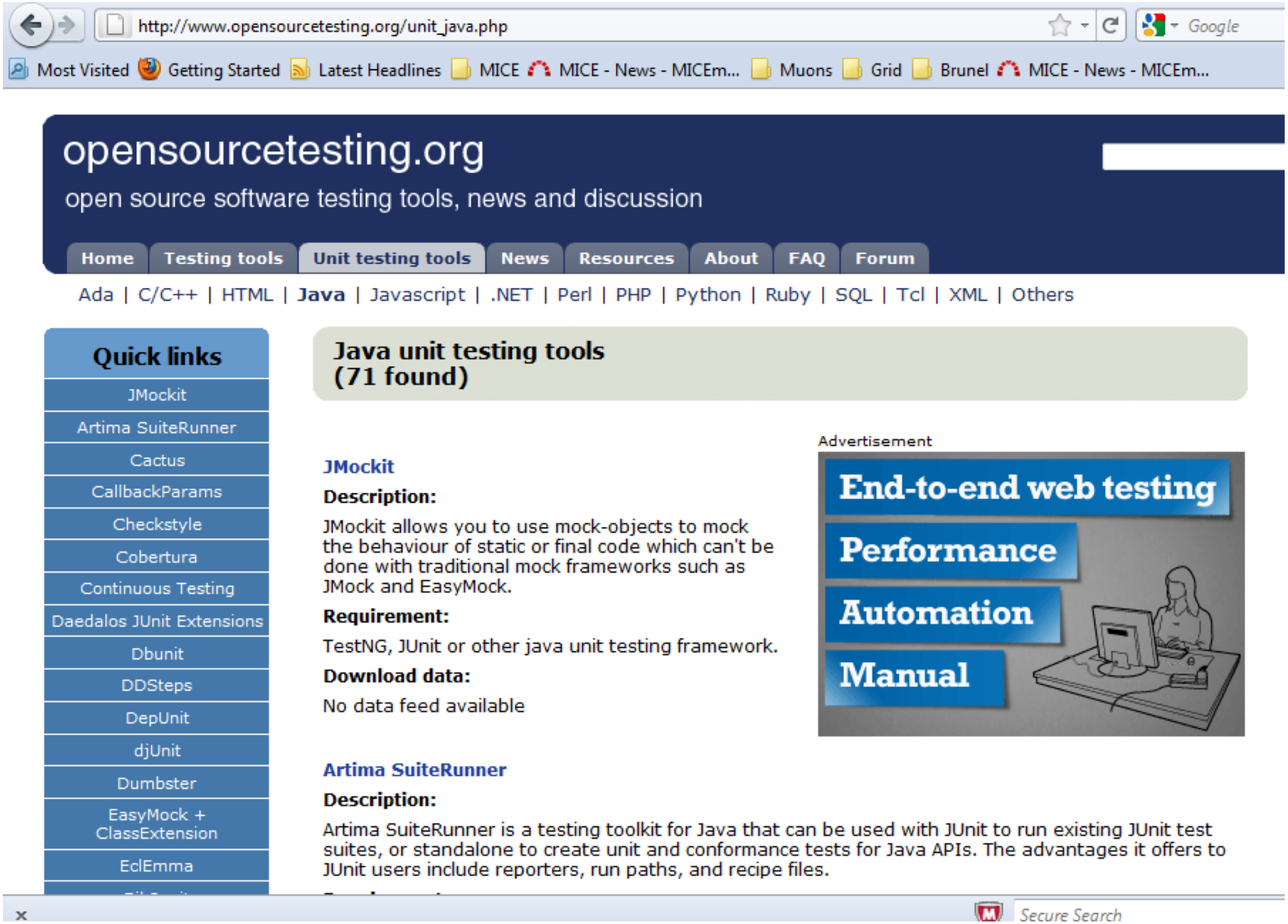
The greater the number of lines covered the harder it is for errors to remain undetected.

At constant test quality improving test coverage, improves error detection

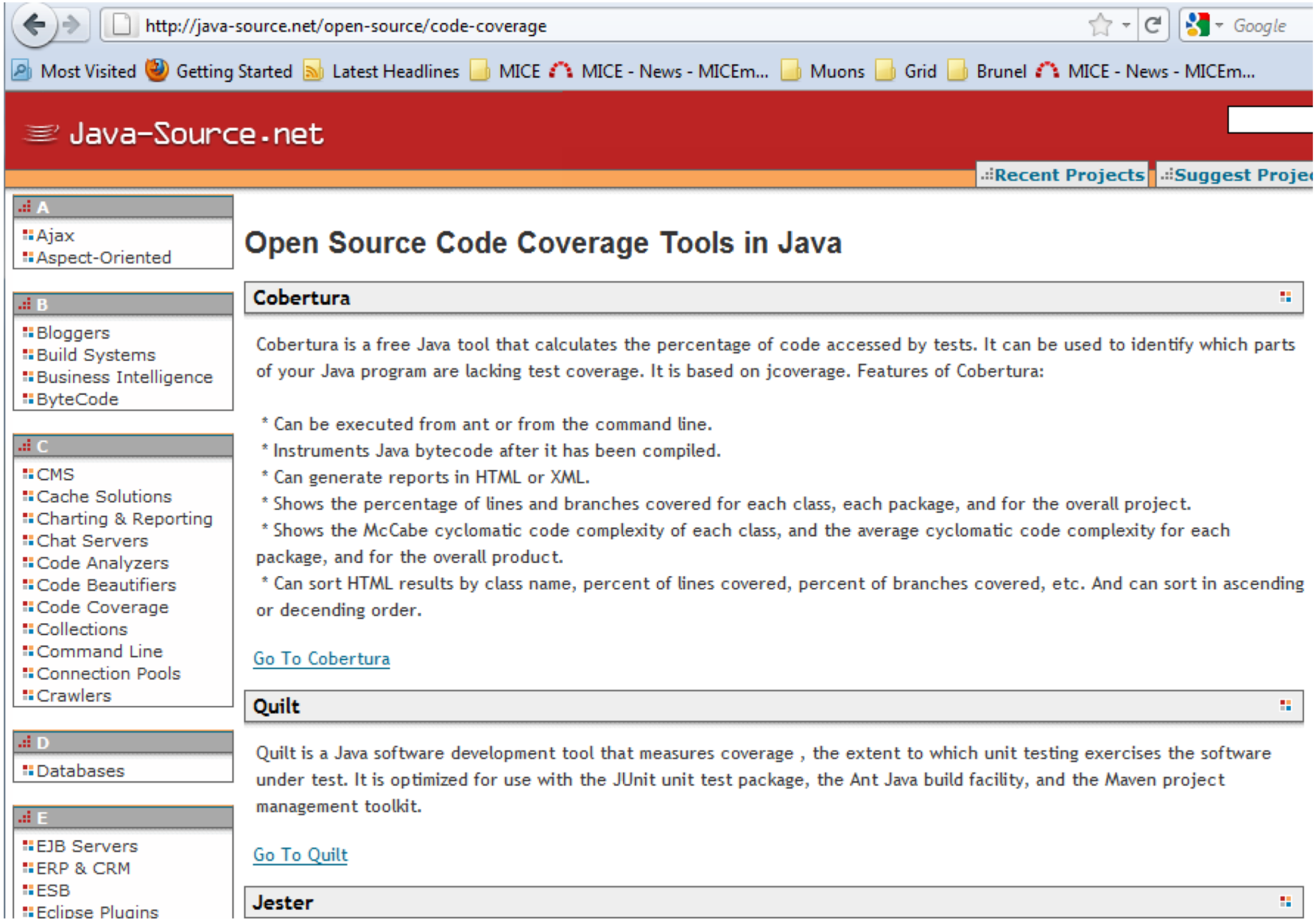
Beware

Bad tests remain bad, even when they cover much of the code.

10 Open Source Unit Tests



11 Open Source Coverage



Improve tests

How can we measure the efficiency of the tests themselves

quis custodiet ipsos custodes?

Mutation testing.

Jester, a package which integrates with Junit. It modifies code (in a way that still compiles) and then reruns the tests.

If it still passes the tests, then the changes are not being tested.

12 Mutation Testing

http://jester.sourceforge.net/

Most Visited | Getting Started | Latest Headlines | MICE | MICE - News - MICEm... | Muons | Grid | Brunel | MICE - News - MICEm...

Jester - the JUnit test tester.

Copyright (2000-2005) Ivan Moore.

- Download
- Documentation
- License
- Example Jester Report
- Hints and Tips
- Bug Reports
- Feature Requests
- Source Repository
- Release Notes

"Why just think your tests are good when you can know for sure? Sometimes Jester tells me my tests are airtight, but sometimes the changes it finds come as a bolt out of the blue. Highly recommended." ***Kent Beck***

Jester finds code that is not covered by tests. Jester makes some change to your code, runs your tests, and if the tests pass Jester displays a message saying what it changed. Jester includes a script for generating web pages that show the changes made that did not cause the tests to fail.

Jester is different than code coverage tools, because it can find code that is executed by the running of tests but not actually tested. Jester's approach is called mutation testing or automated error seeding. However, Jester is not meant as a replacement for code coverage tools, merely as a complementary approach.

Jester is for **Java** code and JUnit tests (download the latest version of Jester). Pester is Jester for **Python** code and PyUnit tests (download Jester version 1.22 and then Pester version 1.01). Nester is a port of Jester for **C#**.

In general, the latest versions of Jester might require the latest versions of Java/JUnit/Python. Older versions of Jester might work with older versions of Java/JUnit/Python. There is absolutely no warranty whatsoever.

There is a really good article by Robert C. Martin and Robert S. Koss on test driven development of bowling scoring code. Running Jester on the code they wrote produced a really unexpected result. To quote Robert C. Martin:

TDD

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

Suitable for black box tests

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

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.

Tests are automatic and self documenting.

TDD improves the code which is written

Not just because code has been tested.

TDD promotes good coding practice

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.

Starting of course with unit tests.

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

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 so 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

Metric	TDD Approach	Traditional Approach
Number of test cases written	629	211
Number of faults detected by the SQA group during all the units tested	74	109
Number of faults detected by the SQA group during integration testing	13	15
Number of faults detected by the SQA group during acceptance testing	14	31
Total Person Hours spent	928	1245

experimental study that was done using undergraduate students at the University of Southern Mississippi. Each of these two groups consisted of 9 students and time period for the whole study was 3 months.

Equality

Object oriented programming methods often create a new object.

That object has state.

How to confirm the state is the correct?

The state should be encapsulated and not accessible from the outside.

Break encapsulation

Provide an equals method.

So determine (by applying the algorithm using pencil and paper) what the state of the object should be and create such an object by hand.

Equality

pencilObject.equals(constructedObject)

can then be tested to be true (or false)

And this can be achieved without breaking encapsulation.

Take care with equals …

Java has an equals method for all objects and be default it just checks that the two objects are the same.

A general method to test for equality can not be provided, so you need to override in particular cases.

Equals properties

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

- It is **reflexive**:
  for any reference value x, x.equals(x) is true.
- It is **symmetric**:
  for any reference values x and y, x.equals(y) should is true if and only if y.equals(x) is true.
- It is **transitive**:
  for any reference values x, y, and z, if x.equals(y) is true and y.equals(z) is true, then x.equals(z) istrue.
- 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()

- hash() returns an integer. Must be consistent 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.

Test infected

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.

Responsibility

For equals method we check that what is passed is suitable.

Not null, an object of the correct type: a complex Number for example

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.

If you pass something which is not a complexNumber the return is not defined.

Could set false for instance

Could throw and exception

Could throw a run time error

If you pass an object pointer to a complex Number

- Return true if real & imaginary parts are equal
- Return false if either is not equal

If you pass an object pointer

- 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

User doesn't need to test

Programming **“by contract”.**

The pre-conditions define what input is acceptable the post-conditions define what actions follow from a call conforming to those pre-conditions.

In our example: if the pre-condition is that the method must be passed a non-null complex number. Then the post-condition is that it will return true if they are equal and false if they are not.

It makes no guarantees as to the response to a message that does not conform to the preconditions.

Return can be true or false. An exception can be thrown or a run-time error which causes a programme failure.

All are acceptable

Contract ii

The programmer decides to allow null pointers of the complex number class to be passed.

Must define what happens in the case that such a pointer is passed.

return false, throw a nullpointer exception.

Requires the method to distinguish the cases.

Either is acceptable

Allows any pointer to be passed.

Much more work in the method, much easier for the user

Implication

Only messages which pass the pre-condition need to be tested.

Thinking about testing leads to decisions been made about the pre-conditions for the method invocation.

Of course if it doesn't occur to you that a user might pass a null, then you won't test for it, or make it a pre-condition that the object pointer must be a complex number.

Testing is not magic. Ineffective tests can be written (like ineffective code).

But it provides a different view and another tool, to improve your technique as well as your code.

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.

UDP v TCP

May get multiple returns to the same sub-problem.

For problems above irrelevant.

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

Empty return is clearly OK

Non return

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.

See chapter on workflow

These complexities need:

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

Normal testing must be followed rigorously.

Conclusion

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