Programming Skills

Software Testing

Unit Testing

Amy Krause

What testing gives us

- Confidence that our software is
  - Correct
  - Robust
  - Scalable
- A way to **safely**
  - Fix bugs
  - Refactor code
- "if it's not tested, it's broken" – bittermanandy, 10/09/2010

Why testing isn't done

- "I don't write buggy code"
- "It's too hard"
- "It's not interesting"
- "It takes too much time and I've research to do"

Why testing is needed

"er….there was a buffer overflow…"

Crash but no crash

- F-16 test flights over Israel

- X = Y / altitude
- Dead Sea = -400m

Image courtesy of David Shankbone

What is Unit Testing?

- Testing smallest possible units of the software
  - E.g., methods, subroutines, classes, modules
  - This is the “unit” of code we will test
- Testing carried out by developer
  - Tests written and performed <u>while</u> implementing
- Tests many possible scenarios
  - Not just the expected case
- A (large) set of small tests
  - Each test builds and executes quickly
  - Don’t have to build your entire application to test one thing

Motivation

- It makes your life easier
  - Writing software is hard
  - Complex system
  - Many dependencies
- Confidence
  - Are you happy you understand the code you just wrote?
  - What about in 6 months time?
  - Have your changes broken the code?
- Encourages good design
  - Code has to be tested in small chunks
  - Gives a more modular piece of software
  - Makes you consider how you want a function to behave

Why developers SHOULD test!

- Finds defects sooner rather than later
  - Quicker and easier to fix
  - Any bug found by a customer is very costly (reputation)
- Refactoring
  - Can simplify routines and tidy code with confidence
- Adding code
  - Have you broken what is already there?
- Live code examples
  - Documentation which you can automatically check is correct

Still hesitating?

- I don’t write buggy code
  - People are naturally very protective of their work
  - Almost all code contains bugs
- It takes too much time
  - More effort up front for code and associated test
  - Less time finding and fixing bugs
- It’s too hard
  - Testing is difficult sometimes
  - Hard test means poor code or design
- It’s not interesting
  - It is no different to the other code you write
  - Easier than the alternative

When to do it

- Right from the beginning
  - Integrated part of the development process
  - Make it part of your routine
- All the time
  - Especially when it looks hard
- Different concepts exist
  - Code-then-test
  - Test-then-code

Code-then-test

- Develop the unit then the test
  - Write the test as soon as you have a piece of functionality
- Pros:
  - Fits closely with the traditional way of developing software
  - More intuitive
- Cons:
  - Focus only on ease of implementation rather than ease of use
  - May end up delaying testing
    - Remember: Test early, test often

Test-then-code

- Test-driven development

- Write the tests, they fail – **RED**
- Write the code to make the tests pass – **GREEN**
- Clean up the code – **REFACTOR**

Pros and Cons

- Pros:
  - Focus on ease of use, not on ease of implementation
  - Testing is an integral part of the development process
  - All code has tests
  - No test gets left behind
  - Naturally leads to small units of code
- Cons:
  - Requires learning a new technique
  - Requires discipline

Unit Testing in Practice

- Let's take a problem and solve it by writing tests first

- Aims and outline:
  - This walkthrough introduces you to using the "test-then-code" approach.
  - The aim is to demonstrate that the *test* is the most important part of the software.

The exercise

- *“Produce a method within a mathematics module that will return the sine of a positive angle between zero and 360 degrees. The input and return type are floats.”*

Analysis

- Let's think about what we want to deliver:
  - a function that will accept numbers between 0 and 360 *only*.
  - a function that will return a floating point number
  - a function that will take the input number, calculate/find the sine of that number.

- Let's think about what we need to test:
  1. The sine calculation of the number is correct
     - *Correctness of execution*
  2. We are passing in a positive number between 0 and 360
     - *Boundary conditions*

Analysis – Correctness tests

- Case 1 (correctness). What do we know about the sine function?
  - *sin*(0)=0
  - *sin*(90)=1
  - *sin*(180)=0
  - *sin*(270)=-1
  - *sin*(360)=0
- So, straight away we have five test cases
  - One should not overlook them as trivial
- These checks belong to the test class
  - Too much for the production code
  - Impact on performance and readability

Analysis – Boundary tests

- Case 2 (boundary conditions)
  - Test feeding in negative numbers and numbers >360
  - Conditional statement at the top of the `MySin` method
  - Throw an exception or return an error/assert message
- If the return value from `MySin` is not the expected value, get the test to output an error message.

Conclusions

- It might seem an unusual way to do something: write a test first and then code later, especially with a "trivial example" like above.
  - How trivial was it after all?
  - Would you get it right first-time?
- However, building up unit tests method-by-method we have a way to:
  - trap errors before they happen
  - prevent the software regressing at all stages of the software life cycle.

Conclusions *(cont)*

- By making the unit tests separate executables, we can:
  - package them together as test suites
  - run them independently from the rest of the code
- Unit tests are an important part of your overall test process
- It's a bit more work and you'll have to persuade *yourself* and others to program like this.
  - The ends justify the hard work
  - A good test-suite can save a lot of rework and prevent a lot of disappointment

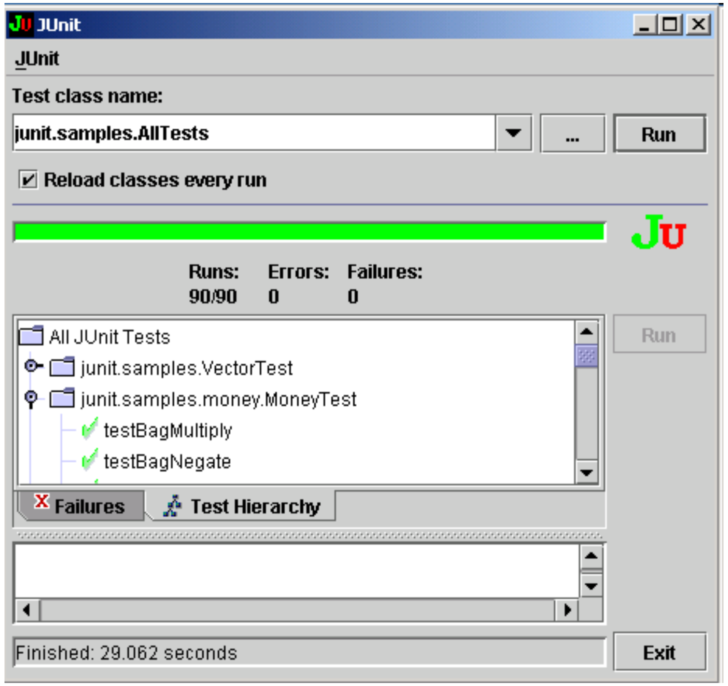
Test Frameworks

- Help automate the process of writing Unit Tests
- Aim is to make Unit Tests:
  - Easy to run
    - One command
  - Fast
    - So they can be run often
  - Repeatable
    - All developers should get the same results
- xUnit refers to a common “style” of test framework

The xUnit Framework

Originally developed by Kent Beck for Smalltalk

- Available for:
  - Java
  - C/C++
  - Fortran
  - Many others
- Two parts:
  - Assertions
  - Test organisation and execution



Assertions

- How do you test your code?
  - Check against expected results
- Assertions
  - Compare actual against expected results
  - Cause the test to pass or fail
  - Encapsulate common checks for convenience
  - Take the form of methods which test arguments for equality
- All tests must assert something!
  - Tests without assertions don't test anything

Testing Java with JUnit

- Using assertions

```
@Test public void testSimpleMath() {
      assertTrue(myMath.isPrime(761));
      assertEquals(5, myMath.add(2,3));
      assertEquals(3.143, myMath.divide(22, 7), 0.001);
}
```

- Testing exceptions (first way)

```
@Test(expected= IndexOutOfBoundsException.class)
public void testEmpty() {
      new ArrayList<Object>().get(0);
}
```

- Testing Exceptions (second way)

```
@Test public void testEmpty2() {
      try {
            ArrayList<Object>().get(0);
            fail("Expected Exception");
      } catch (IndexOutOfBoundsException e) {
            //Expected code flow
      }
}
```

Testing C with CUnit

- Using assertions

```
void test() {
        CU_ASSERT(isPrime(761));
        CU_ASSERT_EQUAL(add(2, 3), 5);
        CU_ASSERT_DOUBLE_EQUAL(divide(22, 7),3.143,0.001);
}
```

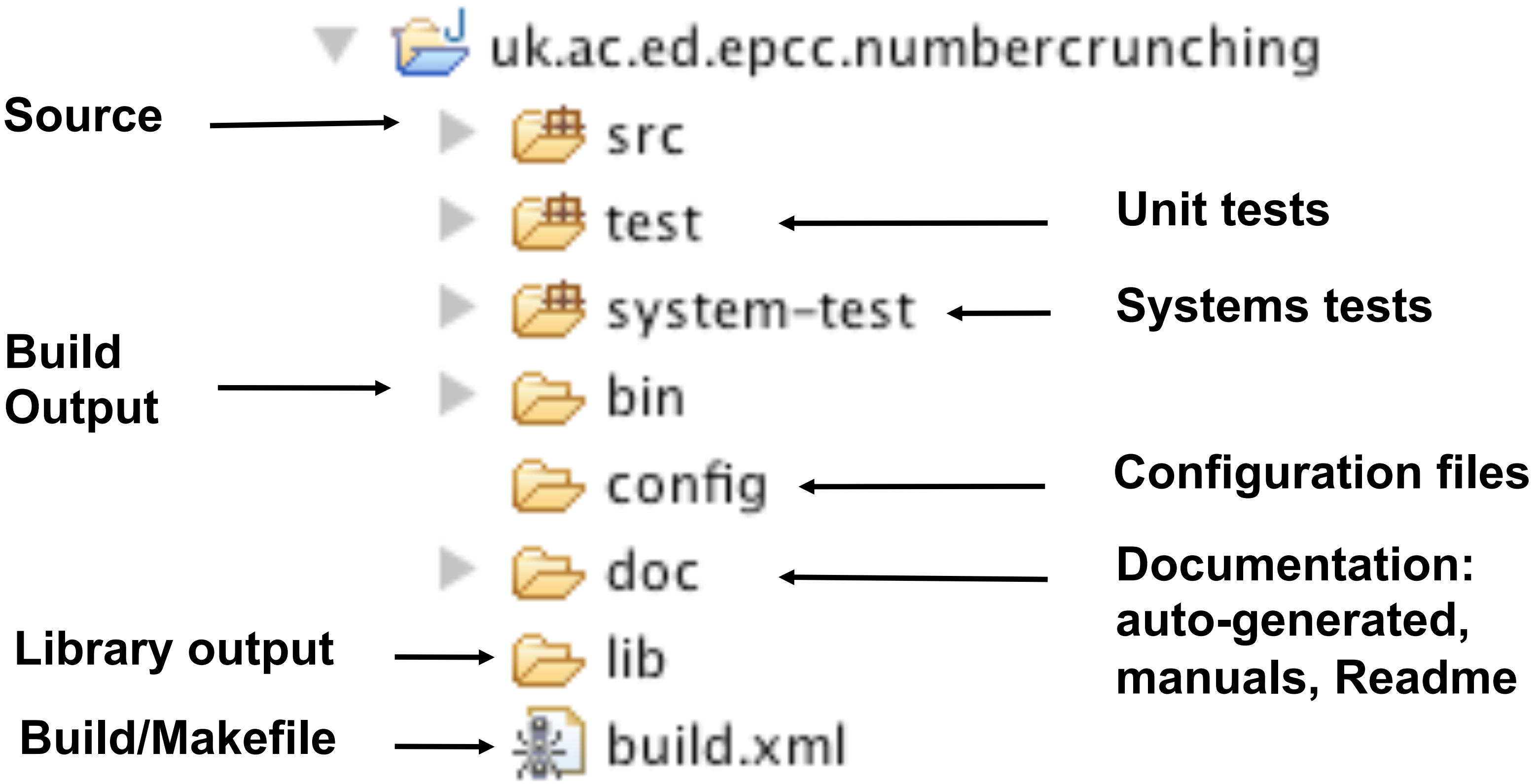
- Testing return codes

```
If (openFile(wrong_name) != ERR_FILE_NOT_FOUND) {
        CU_FAIL("Expected file not found error");
}
```

Test organisation

- Tests normally kept in parallel directory structure to source
  – E.g. `test/myprog/aTest.java` has tests for `src/myprog/a.java`
- Test Suite
  – Contains multiple Test Cases
- Test Case
  – Contains multiple Tests
  – Single setup and teardown method
  – Tests target a specific unit of your code
- Setup/Teardown
  – Contains common code
  – Executed before and after *each* test
- Tests
  – Test a specific aspect of your unit
  – Typically test name starts with “test”
    – `testCarHasDoors`

- For example:


uk.ac.ed.epcc.numbercrunching
src
test
system-test
bin
config
doc
lib
build.xml
Source
Unit tests
Systems tests
Build Output
Configuration files
Documentation: auto-generated, manuals, Readme
Library output
Build/Makefile

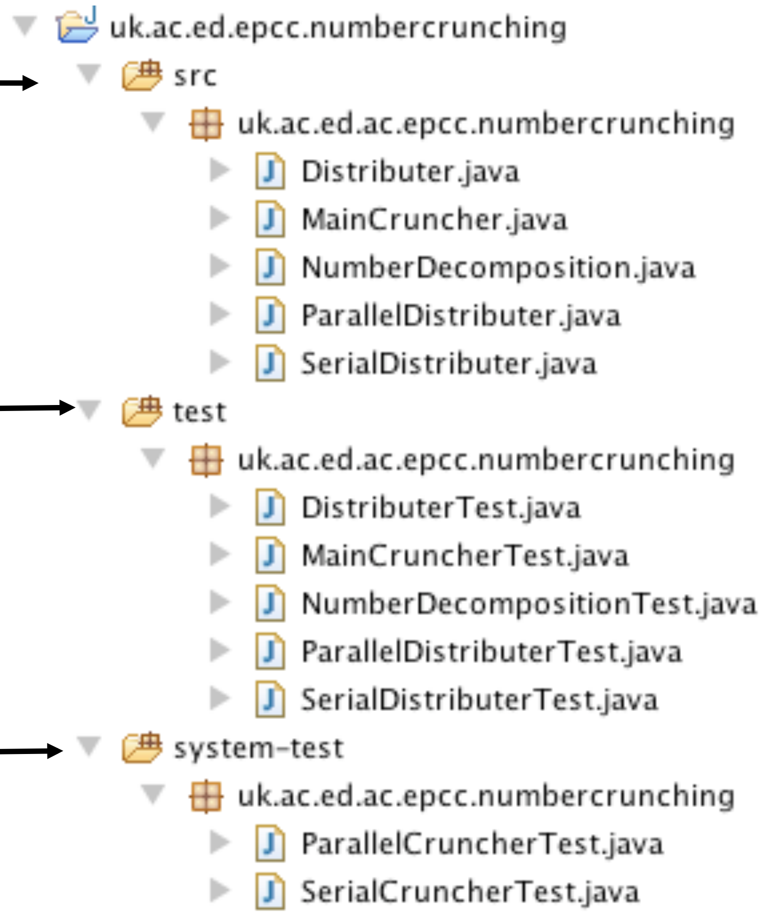
Project structure

- Expanding code folders

Source files

Unit test files
with same package/directory
structure

System test files



Testing Techniques

- Regression Testing
  - Whenever a bug is found, a test is written that exposes it
  - This test ensures that the bug does not reappear in future versions
  - Bug fixes must be tested to ensure they don't introduce new bugs
- Mock Objects and Stubs
  - Classes are not written in isolation
  - But unit tests should only test one class
  - Solution is to mock or stub out dependencies
    - e.g. fake database calls
- Test-Driven Development (TDD)
  - Writing tests before writing code
  - Can help break large problems into small steps
  - Focuses attention on interface to the code

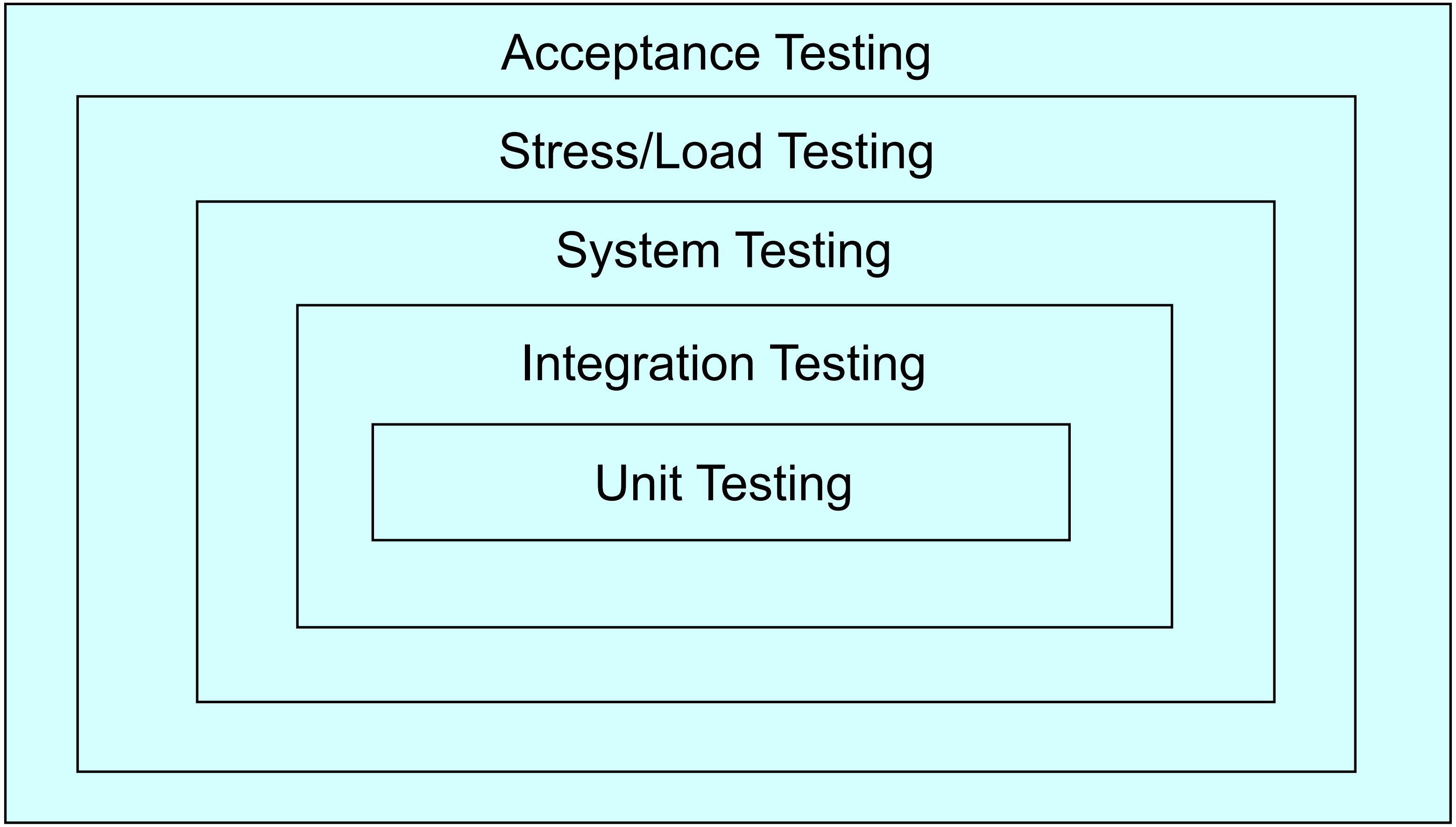
Effects on Code and Design

- Unit Testing and TDD change the design and implementation process
  - Encourage development of small self contained pieces of code
    - Which are easier to test
  - Refactoring and maintenance are much easier
    - Can be confident and aggressive about making changes
  - Encourage thinking about interface design
    - How will the code be called?
  - Unit tests become a resource for other developers
    - Essentially documentation on how to use the code

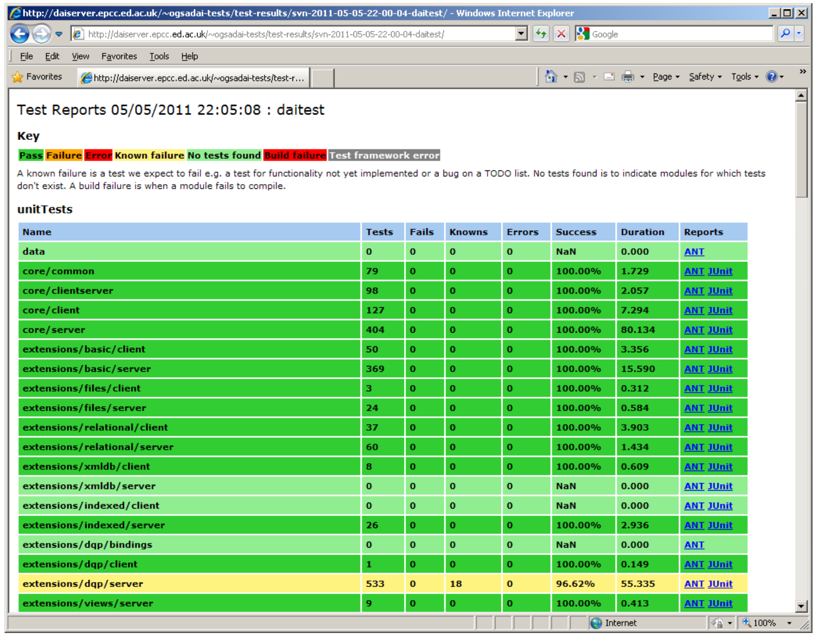
Limitations of Unit Testing

- Bad unit tests will give a false sense of quality
- When projects overrun, testing is the first thing to get squeezed
  - So developers tend to not bother!
- Hard to write unit tests for GUI code
  - Although frameworks are available (Abbot, GUITAR)
- Does not replace other types of testing

Other types of testing



Automated build + test = “fail fast”



Test Reports 05/05/2011 22:05:08 : daitest

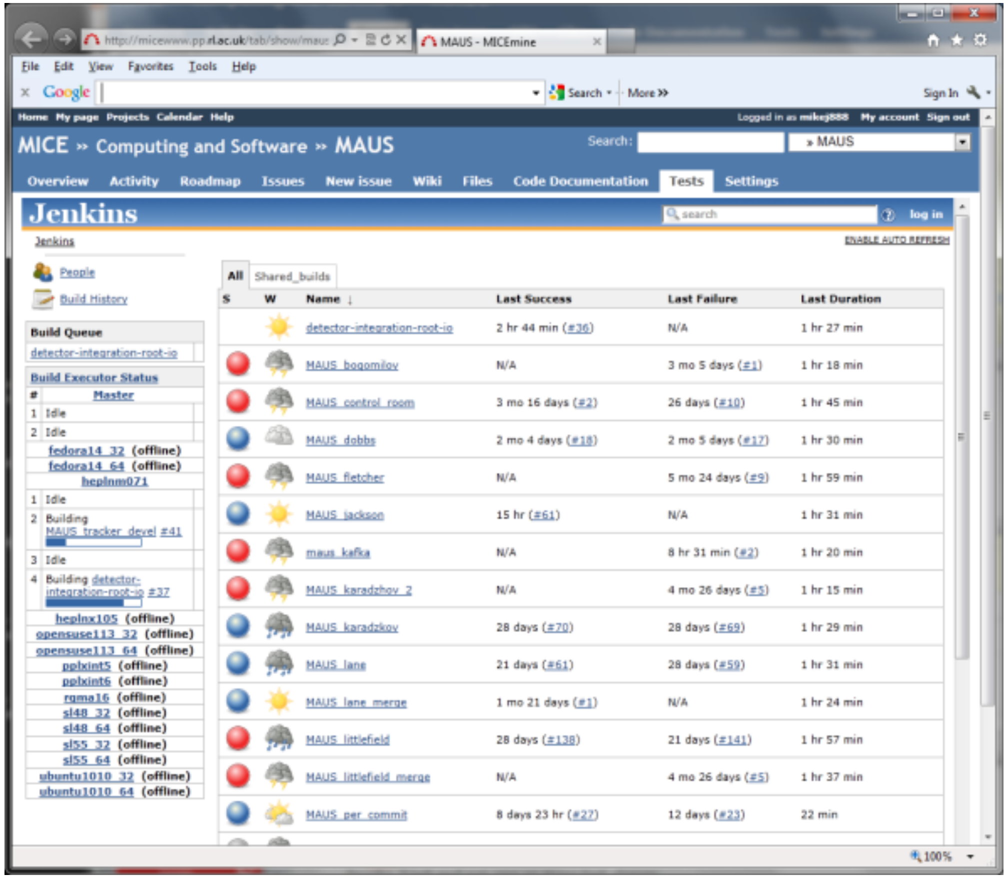
Key

Pass Failure Error Known failure No tests found Build failure Test framework error

A known failure is a test we expect to fail e.g. a test for functionality not yet implemented or a bug on a TODO list. No tests found is to indicate modules for which tests don't exist. A build failure is when a module fails to compile.

unitTests

Name	Tests	Fails	Knowns	Errors	Success	Duration	Reports
data	0	0	0	0	NaN	0.000	ANT
core/common	79	0	0	0	100.00%	1.729	ANT JUnit
core/clientserver	98	0	0	0	100.00%	2.057	ANT JUnit
core/client	127	0	0	0	100.00%	7.294	ANT JUnit
core/server	404	0	0	0	100.00%	80.134	ANT JUnit
extensions/basic/client	50	0	0	0	100.00%	3.356	ANT JUnit
extensions/basic/server	369	0	0	0	100.00%	15.590	ANT JUnit
extensions/files/client	3	0	0	0	100.00%	0.312	ANT JUnit
extensions/files/server	24	0	0	0	100.00%	0.584	ANT JUnit
extensions/relational/client	37	0	0	0	100.00%	3.903	ANT JUnit
extensions/relational/server	60	0	0	0	100.00%	1.434	ANT JUnit
extensions/xmldb/client	8	0	0	0	100.00%	0.609	ANT JUnit
extensions/xmldb/server	0	0	0	0	NaN	0.000	ANT JUnit
extensions/indexed/client	0	0	0	0	NaN	0.000	ANT JUnit
extensions/indexed/server	26	0	0	0	100.00%	2.936	ANT JUnit
extensions/dqp/bindings	0	0	0	0	NaN	0.000	ANT
extensions/dqp/client	1	0	0	0	100.00%	0.149	ANT JUnit
extensions/dqp/server	533	0	18	0	96.62%	55.335	ANT JUnit
extensions/views/server	9	0	0	0	100.00%	0.413	ANT JUnit



Summary

- Unit test code is very useful
  - **Has** to be updated when you add new functionality
  - A unit test which only tests 10% of the unit's code isn't much good
- Tests should be as comprehensive as possible
  - Projects often aim for >95% test coverage (usually line coverage)
  - 100% test coverage practically unachievable
- If writing a unit test gets very complex, maybe your code is too complex
  - Restructure to improve encapsulation and reduce the size
  - Good encapsulation reduces bugs and makes it easier to test

Methods: Summary

- Think about testing at the start of the project
  - Types of testing
  - When will it be done?
  - How long will it take?
- Use Test-Driven Development
  - Write tests, then write code
  - Good unit testing should **save** development time
- Make testing an up-front activity
  - Not an afterthought
  - Test early, test often

Further resources

- Choose a test framework that works for you and your preferred language – for example:
  - CUnit (C)
  - FRUIT (Fortran)
  - CppUnit (C++)
  - Google Test (C++)
  - JUnit (Java)
  - nose (Python)
- Other frameworks for integration testing, for example:
  - REST-assured (RESTful web services in Java)
- Other resources
  - Test Driven Development By Example (Kent Beck)
  - Google Testing Blog: http://googletesting.blogspot.co.uk/

HPC Testing

Motivation

- HPC is hard
  - Code running on large number of processors
  - Complex algorithms
  - Expected results not always known
- Want confidence that (parallel) code is correct
- Do not want to waste time/budget on an HPC machine
- Generate (correct) results quicker

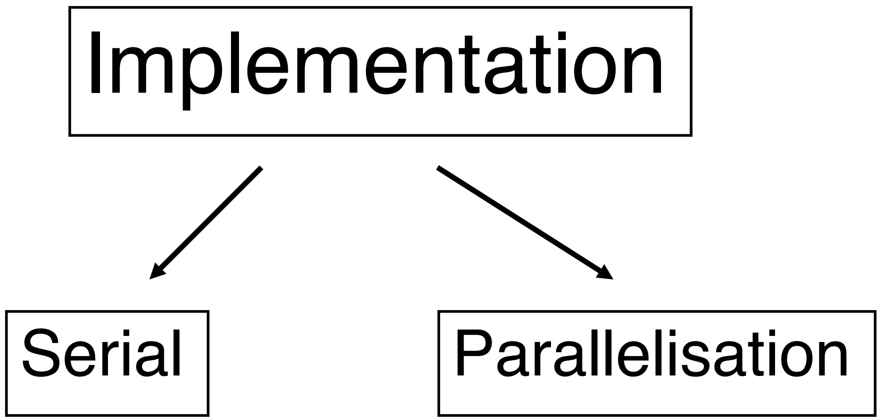
Be Aware Of What You Test

Implementation vs. Algorithm

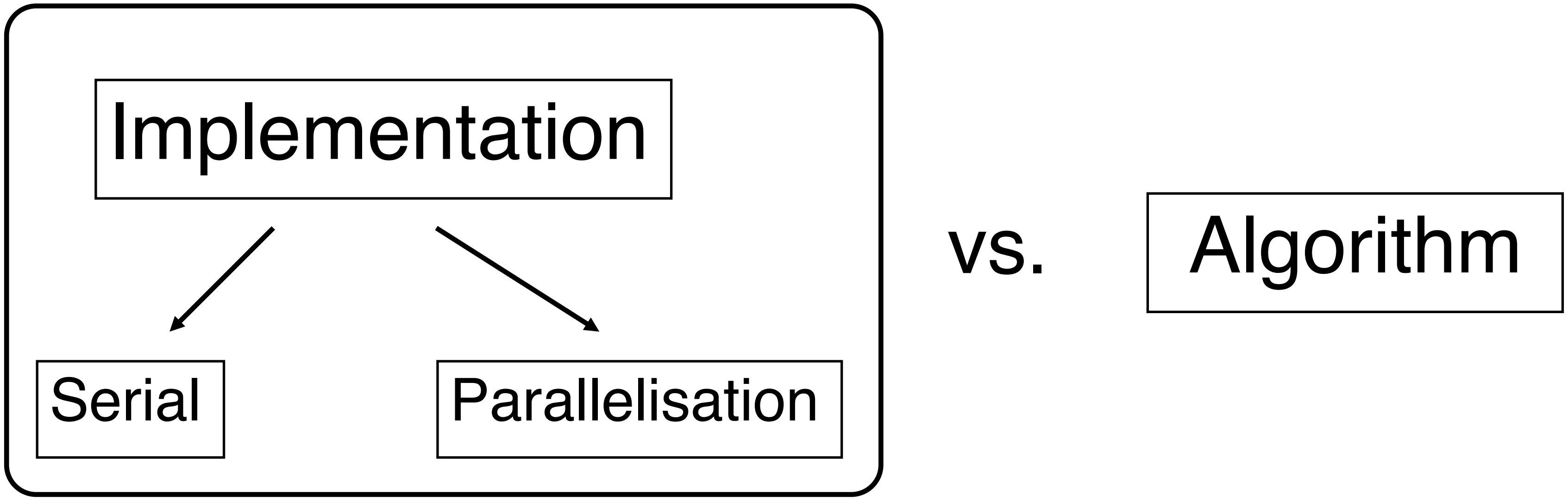
```
r = 0
do i=1, n
  r = r + f(i)
end do
```

$\sum f(x)$

Be Aware Of What You Test



Be Aware Of What You Test



Regression Testing

- Tests how aspects of a program compare to the last run
  - Correctness (unit tests)
  - Performance
  - Program Output
- Unit tests
  - Add a unit test when you fix a bug
  - Notifies you if the bug re-occurs
  - Can be more cumbersome for typical HPC code (do loops)
- Performance
  - Execution time, data transfer speed, memory footprint
  - Track performance changes over time

Regression Testing - Output

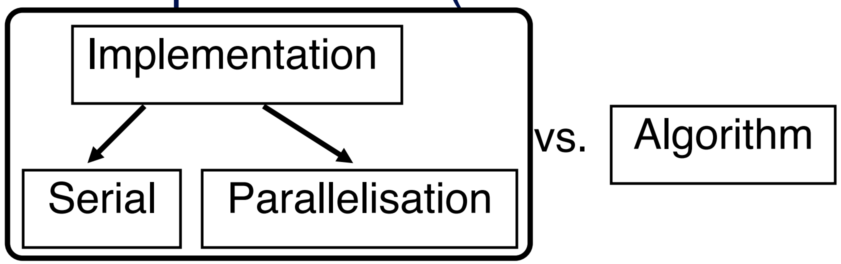
- Comparing the actual output of a program against the expected output

\+ Tests the tested part as a black box
\+ Tests if the last changes broke the code
\+ Difference in output can give clue on what's wrong

\- Does not (necessarily) test individual parts of program
\- Does not always test correctness
  - Expected output might not be correct
  - Test data set may not cover all cases
\- Only tests the program as a whole
\- Usually takes longer than unit tests (hence not executed as frequently)

Useful techniques for HPC (Regression) Testing

- Serial optimisation:
  - Comparing original serial output against output of optimised serial code
- Parallelisation:
  - Compare serial output against output of parallelised version
  - Different data decompositions/number of processors (tests decomposition)
  - Halo swapping verification



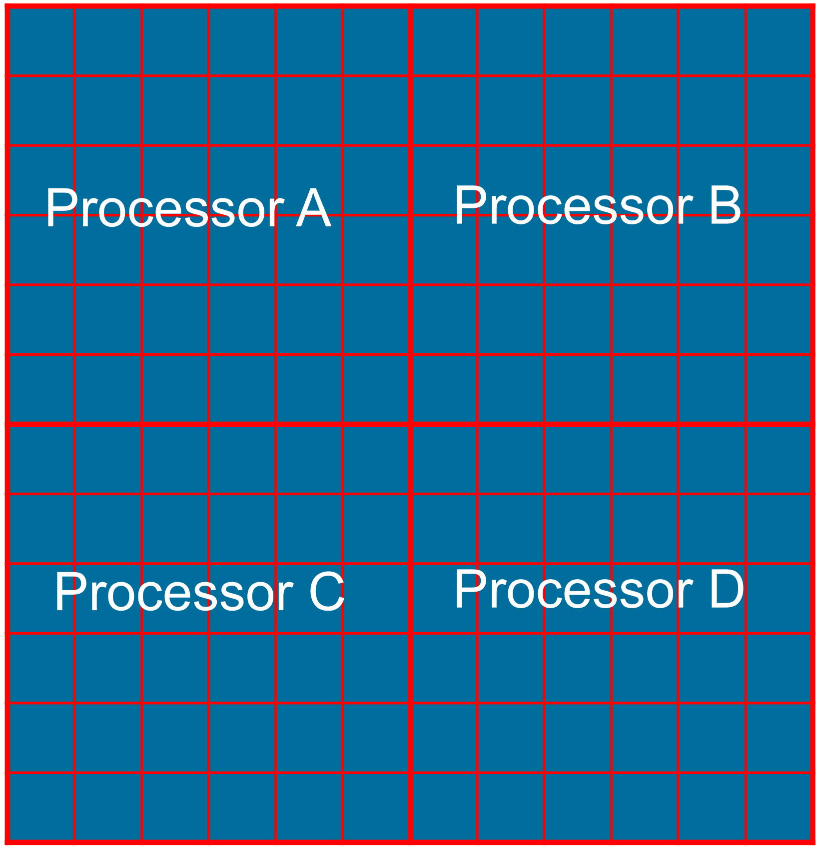
- General:
  - Try to narrow down the problem area
    - Test against single modules
    - Different data sets to test different aspects

Testing Halo-swapping Code

- Parallelisation often involves partitioning data
- Common source of errors

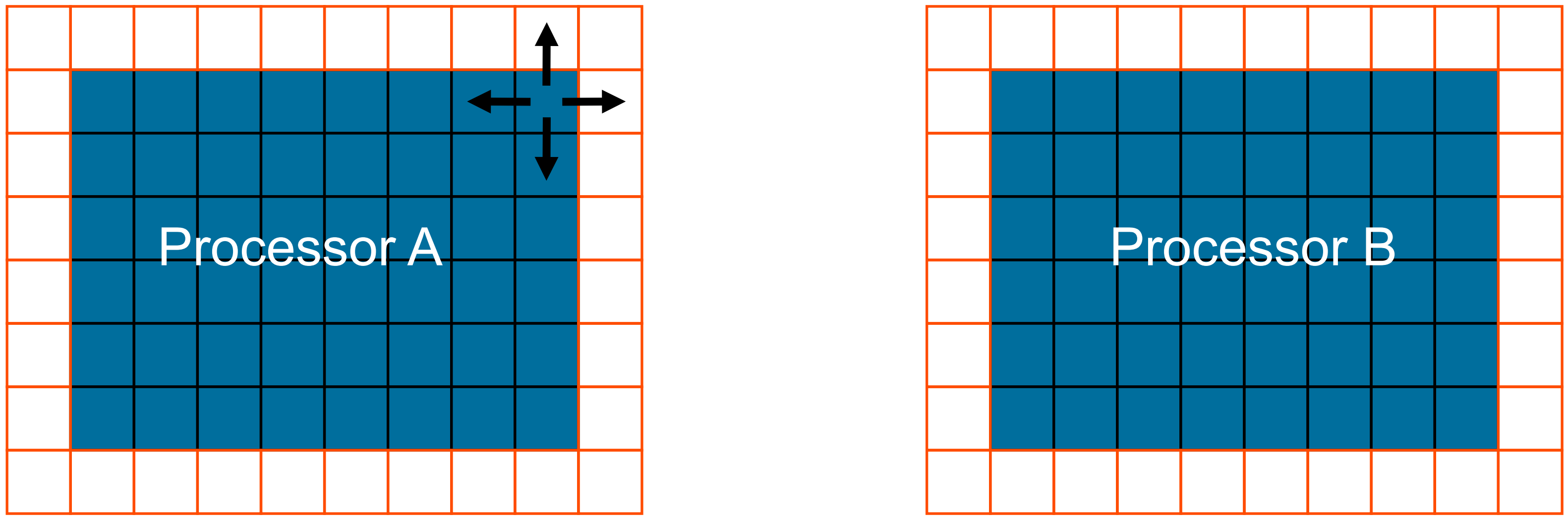
Halo-swapping

- 2D Array decomposed over 4 processors:



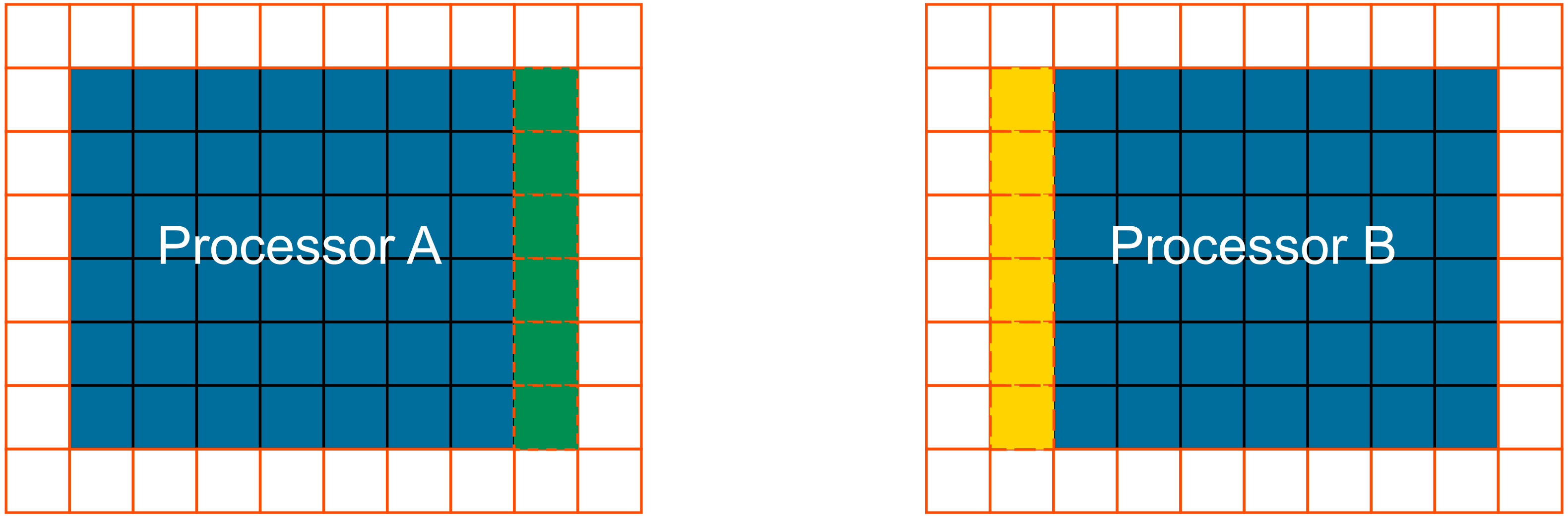
Halo-swapping

- If algorithm needs neighbouring cells:



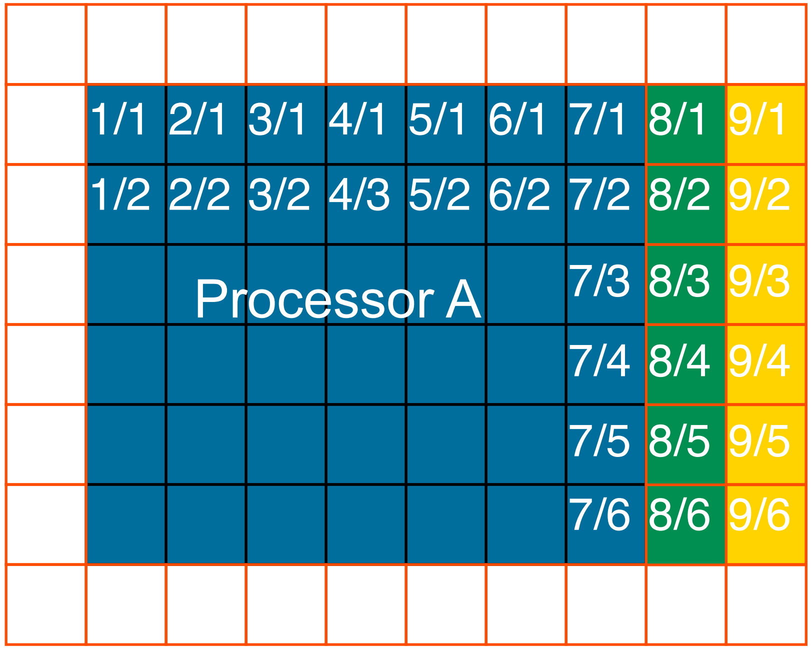
Halo-swapping

- … each processor needs to maintain a copy of surrounding "halo" cells:



Testing Halo-swapping

- Fill cells with co-ordinates; Should be still the same after halo-swapping:



1. Fill your data field with coordinates
2. Execute the Halo-swapping routine
3. Check the coordinates are correct

Floating Point Accuracy

- $A + B \neq B + A$
  - Floating point operations are not guaranteed to be associative
  - Depends on system architecture
  - Compiler

- Optimised code can be different
  - Compiler optimisation too aggressive?

- Does the optimised version have to be identical?
  - What is an acceptable tolerance?
  - Measure equality using this tolerance in your tests

Design Considerations

- Separate your code into testable modules
  – IO routines
  – Decomposition
  – Halo-swapping
  – Algorithm

- Serial vs. Parallel
  – Allow easy switching between serial and parallel versions
  – Help find the source of bugs quickly
    – Not in the serial version? Suggests a parallelisation problem

Common problems

- Random numbers and parallel code
  - More than one Random Number Generator in parallel code
  - Different results when using different numbers of processors
- Global sum
  - The order of calculations can matter
  - Different decomposition means different ordering
  - Implement the global sum independent of the decomposition
- Floating Point
  - Single vs. Double precision
  - 32 vs. 64 bit
  - Small errors can be multiplied in large calculations
  - System architecture

Testing Algorithm Implementation

- Symmetry Testing Example: A ball falling in a vacuum
  - Run the simulation for 10 seconds
  - The ball has a new position and velocity
  - Reverse the velocity
  - Run the simulation for 10 more seconds
  - The ball should be where it started

Conclusion HPC Testing

- Hard, but not impossible
- Be aware of what you test
  - Algorithm
  - Implementation
  - Parallelisation
- Use regression testing to spot bugs early
- Test your parallelisation
  - Decomposition
  - Number of processors
  - Halo swapping
- Write your code with tests in mind
  - Modularise