

## Testing

*David Croft*

Introduction

Testing

Unit testing

Integration testing

System testing

Acceptance testing

How to...

Unit test

Automate

Recap

# Testing

David Croft

Coventry University

david.croft@coventry.ac.uk

March 19, 2018

## 1 Introduction

## 2 Testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing

## 3 How to...

- Unit test
- Automate

## 4 Recap

How many bugs in a 1000 line program?

- Industry average 15-85 per KLOC.
- 1 KLOC (Kilo Lines Of Code) == 1000 lines of code.

How many lines of code in something like Office?

- Libreoffice has 12.5 million lines of code.
- Between 6,250 and 37,500 bugs.

How many make it through to the customer?

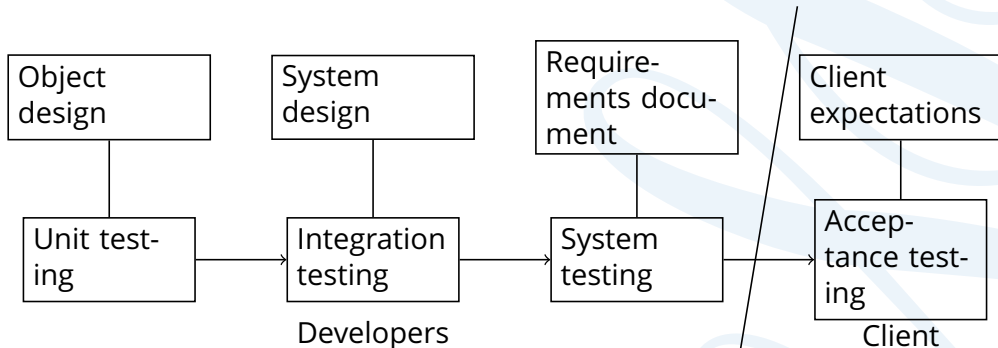
- 0.5-3 per KLOC.
- How do we get it down to that?
  - We test, a lot.

*"If I write good code it won't have bugs."*

–Every programmer ever

Your code will have bugs.

- The gold standard for perfect code belongs to.....The Space Shuttle.
- 420,000 lines of code.
  - Expect between 210 and 720 bugs.
- In 1996 the previous 3 versions had one known bug each.
  - 0.0024 per KLOC.
- Research in own time if interested.



Ad-hoc  
Acceptance  
Accessibility  
Agile  
API  
Automated  
All Pairs  
Beta  
Black Box  
Backward Compatibility  
Boundary Value

Bottom up Integration  
Branch  
Compatibility  
Component  
Condition Coverage  
Dynamic  
Decision Coverage  
End-to-end  
Exploratory  
Equivalence Partitioning  
Functional

GUI  
Glass box  
Gorilla  
Happy path  
Integration  
Interface  
Internationalization  
Keyword-driven  
Load  
Localization  
Negative

Pair  
Performance  
Penetration  
Regression  
Risk based  
Smoke  
Security  
Sanity  
Scalability  
Stability  
Static

System  
Soak  
System Integration  
Unit  
Usability  
User Acceptance  
Volume  
Vulnerability  
White box

Testing is not just about code.

- Testing expectations, documentation.
- Testing assumptions.

Absence of evidence is not evidence of absence.

- Just because you can't find the bugs doesn't mean they aren't there.
- Formal verification is the exception.
  - Mathematical proof of correctness.
  - Mathematical model of an algorithm.
  - The maths proof is much harder than writing the code.

Once you've written your code, what is the most important step?

- Testing happens continuously during development.
  - I.e. does my code compile/run/work?
- Important to do formal testing
  - Just checking it runs as you code is not enough.
  - Make sure you've not missed anything
  - In depth, comprehensive testing.
- Extra attention to edge cases.
  - I.e. if code expects number between 0 and 100 make sure to test -1, 0, 1, 99, 100 and 101.
- Every return path.
  - I.e. every if-else.

Testing is important. Bugs can kill.

Your bugs not that serious but still have a duty to do reasonable testing.

- 2005 Toyota Camry acceleration issues.
  - Multiple crashes, injuries and death.
- Code was reviewed afterwards by experts but not released.
  - Michael Barr & Phillip Koopman.
- Described as untestable “spaghetti code”.
  - More than 10 000 global variables.
  
- Bug in the Therac-25 radiation therapy machine.
- Multiple patients received massive overdoses.
  - Patients suffered skin grafts, loss of body parts and/or death.

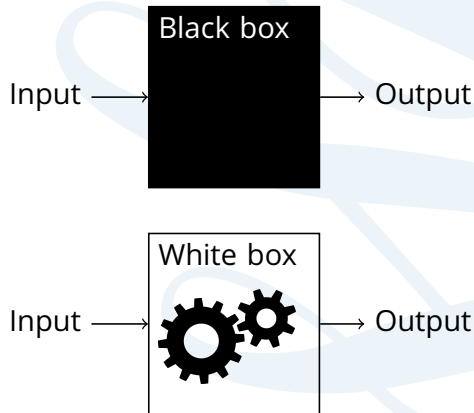


Test each individual 'unit' of your program.

- Python/C++ lets you break your code into modules
  - import/include modules
- Test each module separately
  - Everything module can do
  - Works correctly.
  - Fails correctly.
- Can be white or black box.
  - White box - know/care how module works inside.
  - Black box - don't know/care how module works inside.
- Version control is great help here.
  - Multiple programmers working on separate units.
  - Commit code only if it passes unit testing.

## Black and white box testing

- Black box.
  - Don't see/know what's going on inside.
  - Just supply inputs, test outputs.
- White box.
  - Do see/know what's going on inside.
  - Test internal states/variables.



Test how multiple modules/units work together when combined.

- Individual modules treated as black boxes.
  - Don't care how they work.
  - Just care what they do.
- Make sure everyone is following agreed interfaces.
  - Function names/parameters etc.
  - Behaviour hasn't changed.
- Continuous integration.
  - Bring together everyone's latest code several times a day.



Test system meets the specifications.

- Test the whole system works together.
- Black box testing.
- Ideally done by someone other than the developer/s.



Not testing code directly.

- Testing expectations.
- Does the whole thing work as expected?
  - Were specifications correct?
  - Were specifications complete?
- The customer is not always right.
  - Just because they wrote the specifications doesn't mean they meant it.
  - "Knowledge extraction" is hard.

## Testing

David Croft

Introduction

Testing

Unit testing

Integration testing

System testing

Acceptance testing

How to...

Unit test

Automate

Recap

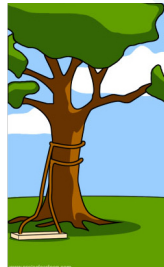
# Expectations



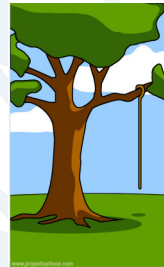
How the customer explained it



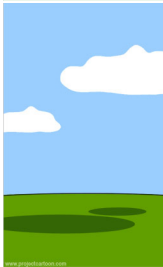
How the project leader understood it



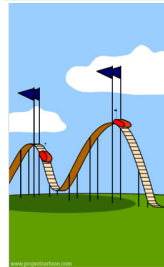
How the programmer wrote it



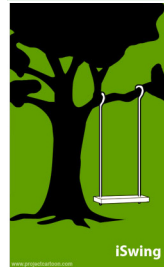
What operations installed



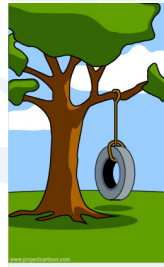
How the project was documented



How the customer was billed



What marketing advertised



What the customer really needed

# How to Unit test your code I



Just looking at unit testing in 122COM.

Good unit testing is very time consuming.

- Should be testing before committing any changes.
  - New feature? unit test
  - Bugfix? unit test
  - Code re-factoring? unit test
  - Bored? unit test

Why bother??

- Debugging is simpler, know where bugs are.
- Bugs stay dead (or detected).
  - Spot new bugs.
- Every 3 bugs solved creates 1 new one (Glenford Myers - Art of Software Testing).

## Basic unit testing.

- Grab your spreadsheet.
- Example - testing your stack code from data structures week.

ID	Description	Test	Expected	Success	Why
1a	Push to empty stack	.push('A')	.size() =1, .top() = 'A'	Pass	
1b	Push to full stack	.push('Z')	StackFull exception	Pass	
1c	Push to !full !empty stack	.push('Q')	.size() += 1, .top() = 'Q'	Pass	
2a	Pop from empty stack	.pop()	StackEmpty exception	Fail	No exception raised
2b	Pop from full stack	.pop()	.size() -= 1, .top() = element at .size()-1	Pass	
2c	Pop from !full !empty stack	.pop()	.size() -= 1, .top() = element at .size()-1	Pass	

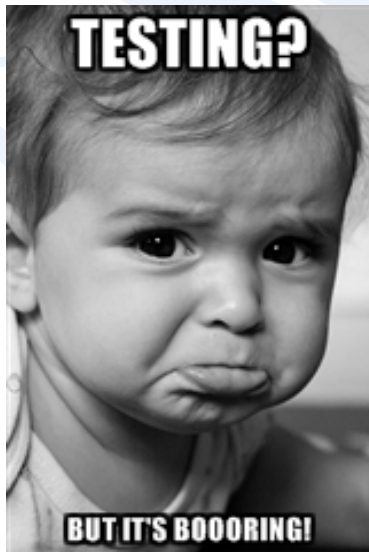


Running unit tests manually is a massive time sink.

- Solution?
  - Automate our testing.
  - Write code to test our code.

I don't wanna

I



Already encountered this idea in this module.

- C++ intro, searching, SQL, data structures and sorting labs.
  - Had code to automatically test your code

Advantages.

- Tested your code.
  - Every time you ran it.
- Quickly tested your code

Disadvantages.

- Messy, confusing testing code.
- Results not clear.
- You guys kept finding ways to break my testing code.

## Solution?

- Unit testing libraries.
- Available for every significant language I can think of.
  - Multiple libraries per language.
- Same concept
  - Write small test functions.
  - Run them all.
  - Report what failed and summary.

## Using unittest module.

- Built in.
- Test ways things are correct.
- Test that things go wrong.
  - Test for expected exceptions.

```
import unittest

class Tests(unittest.TestCase):
    def test_bigger(self):
        self.assertTrue( 1 < 0 )

    def test_equals(self):
        self.assertEqual( 1+1, 2 )

    def test_div(self):
        with self.assertRaises(ZeroDivisionError):
            1 / 0

if __name__ == '__main__':
    unittest.main()
```

lec\_unittest.py

```
F..
```

```
=====
```

```
FAIL: test_bigger (__main__.Tests)
```

```
-----
```

```
Traceback (most recent call last):
```

```
  File "lec_unittest.py", line 5, in test_bigger
```

```
    self.assertTrue( 1 < 0 )
```

```
AssertionError: False is not true
```

```
-----
```

```
Ran 3 tests in 0.000s
```

```
FAILED (failures=1)
```

## Using cxxtest.

- Very similar to Python unittest.
- Slightly more complicated to run.
- Header file, .h file.

```
#include <cxxtest/TestSuite.h>

class SomeTests : public CxxTest::TestSuite
{
public:
    void test_bigger()
    { TS_ASSERT( 1 < 0 );
    }

    void test_equals()
    { TS_ASSERT_EQUALS( 1+1, 2 );
    }

    void test_except()
    { TS_ASSERT_THROWS_ANYTHING( throw 1 );
    }
};
```

lec\_unittest.h



```
Running cxxtest tests (3 tests)
```

```
In SomeTests::test_bigger:
```

```
lec_unittest.h:8: Error: Assertion failed: 1 < 0
```

```
..
```

```
Failed 1 and Skipped 0 of 3 tests
```

```
Success rate: 66%
```

## Python

- Just run it.

```
$ python3 -m unittest TESTCASES.py
```

If you have `unittest.main()`

```
$ python3 TESTCASES.py
```

## C++

- Generate a 'runner' that will actually run the tests.

```
$ cxxtestgen -error-printer TESTCASES.h -o runner.cpp
```

- Compile the runner.

```
$ g++ -std=c++14 -I. runner.cpp -o runner
```

- Run the runner.

```
$ ./runner
```





## Running multiple tests.

- Will have lots of commonalities.
- Each test run on fresh structure.
  - I.e. testing stack/queue
- Have to create/clean up structure for every test.
  - Is a hassle.
- Built in feature to do it for you.
  - `setUp()`
  - `tearDown()`

```
from lab_stack import *
import unittest

class StackTest(unittest.TestCase):
    def setUp(self):
        self.testvalues = 'abcde'
        self.s = Stack( len(self.testvalues) )

    def tearDown(self):
        pass

    def test_size(self):
        """ test that stack reports the correct number of things
        ↪ on the stack """
```

Can be integrated into projects in many ways.

- Build scripts - every time you compile, tests run automatically.
- Commit tests - every time you try and commit a new version, tests run automatically.
- Reports - automatically generate reports on current bugs, track progress.

## Testing

*David Croft*

Introduction

Testing

Unit testing

Integration testing

System testing

Acceptance testing

How to...

Unit test

Automate

Recap

# Quiz

# Why do I care?

## Everyone

- If you don't test your code you will have bad code.
- Testing is a vital employability skill.
- Automated testing is a very useful employability skill.
  - Automated testing can save you time.

- Unit test - test individual 'units' of code.
  - Functions, classes etc.
- Integration test
  - Test multiple units work correctly when combined.
- System test.
  - Test the whole thing matches what the user said they wanted.
- Acceptance test.
  - User/s test what they said they wanted is what they actually wanted.
- Automated unit testing.
  - What is it?
  - Why do it?
  - How to do it.

## Testing

*David Croft*

Introduction

Testing

Unit testing

Integration testing

System testing

Acceptance testing

How to...

Unit test

Automate

Recap

# The End