Testing

David Croft

Coventry University

david.croft@coventry.ac.uk

March 4, 2016

Overview

Coventry University

Bugs

A

How many bugs in a 1000 line program?

- Industry average 15-85 per KLOC.
    - 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 so low?
    - Testing.

Bugless

A

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

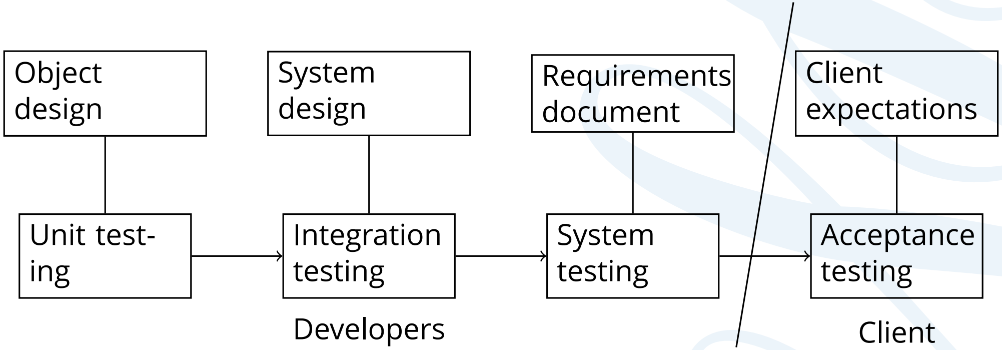
–Every programmer ever

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 version had had one known bug each.
  - 0.0024 per KLOC.

Testing types

C



Ad-hoc	Bottom up Integration	GUI	Pair	System
Acceptance	Branch	Glass box	Performance	Soak
Accessibility	Compatibility	Gorilla	Penetration	System Integration
Agile	Component	Happy path	Regression	Unit
API	Condition Coverage	Integration	Risk based	Usability
Automated	Dynamic	Interface	Smoke	User Acceptance
All Pairs	Decision Coverage	Internationalization	Security	Volume
Beta	End-to-end	Keyword-driven	Sanity	Vulnerability
Black Box	Exploratory	Load	Scalability	White box
Backward Compatibility	Equivalence Partitioning	Localization	Stability	
Boundary Value	Functional	Negative	Static	

More than bugfixing

C

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.

- Mathematically proof of correctness.
- Mathematical model of an algorithms.
- Can still mess up the code.

Testing

C

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

- Testing happens continuously during development.
  - Code compiles?
  - Code runs?
  - Code works?
- 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.

Unit testing

C

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.

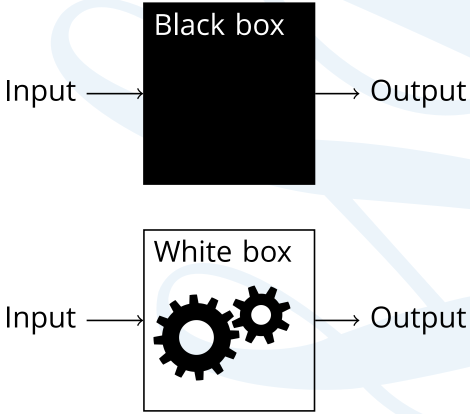
Coventry University

Black box White box

C

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.



Integration testing

C

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.

System testing

I

Test system meets the specifications.

- Test the whole system works together.
- Black box testing.

- Ideally done by someone other than the developer/s.

Acceptance testing

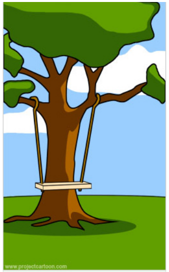
Not testing code directly.

- Testing expectations.

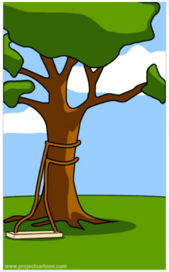
- Does the whole thing work as expected?
  - Were specifications correct?
  - Were specifications complete?

Expectations

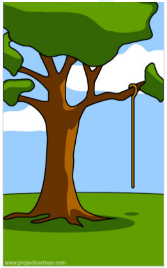
www.projectcartoon.com
How the customer explained it



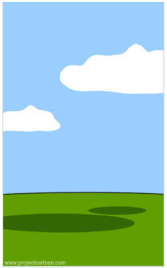
How the project leader understood it



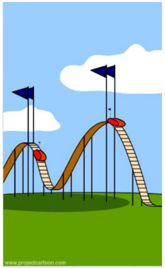
How the programmer wrote it



What operations installed

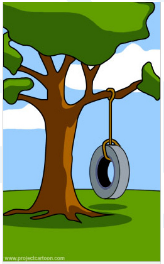


How the project was documented



How the customer was billed

iSwing
www.projectcartoon.com
What marketing advertised



What the customer really needed

How to Unit test your Dragon I

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).

How to Unit test your Dragon II

Basic unit testing.

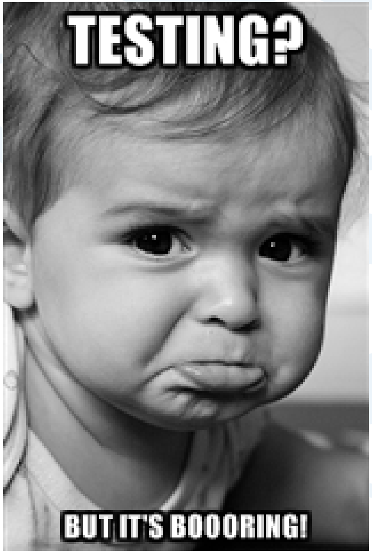
- Grab your spreadsheet.
- Example - testing your stack code from data structures week.

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	

I don’t wanna

Running unit tests manually is a massive time sink.

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



Automated unit testing

C

Already encountered this idea in this module.

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

Advantages.

- Fully tested your code.
  - Every time.
- Quickly tested your code

Disadvantages.

- Messy, confusing testing code.
- Results not clear.

Automated unit testing libraries

C

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.

Python testing

Using unittest module.

- Built in.

- Test was things are correct.
- Test that things go wrong.
  - Test for expected exceptions.

```python
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

Python testing results

I

```
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)
```

C++ testing

Using cxxtest.

- Very similar to Python unittest.
- Slightly more complicated to run.

```
#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

C++ testing results

```
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%
```

Running the tests

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++11 runner.cpp -o runner
```

- Run the runner.

```
>> ./runner
```

putUp, tearDown

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 hassle.
- Built in functions to do it for you.
  - `setUp()`
  - `tearDown()`

putUp, tearDown II

```
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 """
```

Advanced tricks

A

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.

Quiz

Recap

- 🟢 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.

The End