Systematic Testing

Murray Crease
Scott Logic

Introduction

- About Scott Logic
- What is Testing?
- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing
- Performance Testing
- Usability Testing

Scott Logic

> Bespoke Software Development
- Clients across the world
- Work primarily out of Scott Logic offices

> Range of Services
- Development
- UX
- Project Management
- Testing

> All about the technology
- A software house not an IT department
- Smart people, varied technologies

Scott Logic in Edinburgh

- Central location
  - Castle view (ish)!
- 38 staff
  - 31 Developers
  - 4 UX
  - 3 Management & Admin
- Range of Technologies
  - Java, C# .NET, JavaScript/HTML5
- Range of Social Activities
  - Squash, Badminton, Running
  - Xbox, board games, pub quiz
  - Karting, Go Ape, white water rafting.

What Is Testing?

> It involves the execution of a software component or system to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

http://en.wikipedia.org/wiki/Software_testing

Meets Requirements

- Users / Business Owners know what they want and can enumerate
- Testers check software meets these requirements

- List of requirements -> list of tests -> list of passed/failed

- Users rarely know what they want
  - And even more rarely are able to enumerate

Can Be Installed and Run

- Makes sense … need to build it first

Responds Correctly To Inputs

> Know what the possible inputs are
  - Test them all

> Could be a large number of inputs
> In arbitrary combinations

> Almost an infinite set

Runs in An Acceptable Time

- Where acceptable = ???
- For what inputs?
- Scalability

Sufficiently Usable & Desired Functionality

> Hmmmm??

Why is Testing Actually Good?

> Money
- Software bugs cost global economy $312 Billion annually (CU)
  > ~10% UK GDP

> Functionality
- Software needs to perform

> Reputation
- Buggy software impacts reputation

When Should Software Be Tested?

> Earlier is better
- Avoid building on top of bugs

> Earlier Detection is Cheaper (Boehm)
- Resolving bug after delivery 100x more expensive than before
  > 5x for simple systems
  > Good design mitigates this cost

> Cost of correcting bugs during coding is (Type Mock)
- 10% the cost of correcting bugs during QA which is
- 50% the cost of correcting bugs after release

Aside – Agile not Waterfall

> Waterfall

- Build system then test it
- Whole system could be built on false premise
- Users no visibility until completion

> Agile

- Testing built in from start
- Short iterations of coding AND testing
- Users have early & frequent visibility of solution

Creating ‘Test’ Environments

- Typical enterprise setup has three different environments:
  - Development
  - Beta/Stage/UAT
  - Production
- All environments should be the same
  - Resources/Dependencies
  - Only differences in configuration (e.g. DB connection strings)
  - Promotions should be reproducable
- All changes progress through environments in order
  - Dev -> Beta -> Production
- Data can be copied in reverse direction
  - Typically Production -> Beta

Development Environment

> Volatile
- Developers continuously changing
- Can be updated automatically via build server

> Latest changes
- Partial functionality
- Shouldn't be (but maybe) broken

> This is different from the developer machines
- A 4th, personal environment

Beta Environment

> Typically has release candidates
  - System Testing

> Managed promotions
  - Usually by developers
  - Dry run of production release

> Occasionally a separate environment for User Testing

> Code / DB changes promoted from Dev

> Data can be copied from Production
  - Issues around changes in DB promoted from Dev

Production Environment

> The live system

> Managed releases

- Little/no developer involvement
- Rollback
- Typically outside working hours

Levels of Testing

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Unit Testing

> Written by developers on units of code
  - Considers correctness only of isolated code

> Drives design of code
  - Decoupling
  - Modularity

> Automated

> Tools & Frameworks

Unit Testing – Code Design

```
public List<Person> getPeople(String surnamePrefix)
{
	Connection connection = new DatabaseConnection("MyDatabase");
	connection.execute("select * from People where surname.Like('%' + surnamePrefix);

	List<Person> people = new ArrayList<Person>();

	for(String[] components : connection.GetResults())
	{
		Person person = new Person();
		person.firstName = components[0];
		...

		people.Add(person);
	}

	return people;
}
```

Unit Testing – Code Design

```
public List<Person> getPeople(String surnamePrefix)
{
        Connection connection = new DatabaseConnection("MyDatabase");
        connection.execute("select * from People where surname.Like('%' + surnamePrefix);

        List<Person> people = new ArrayList<Person>();

        for(String[] components : connection.GetResults())
        {
                Person person = new Person();
                person.firstName = components[0];
                ...

                people.Add(person);
        }

        return people;
}
```

Unit Testing – Code Design

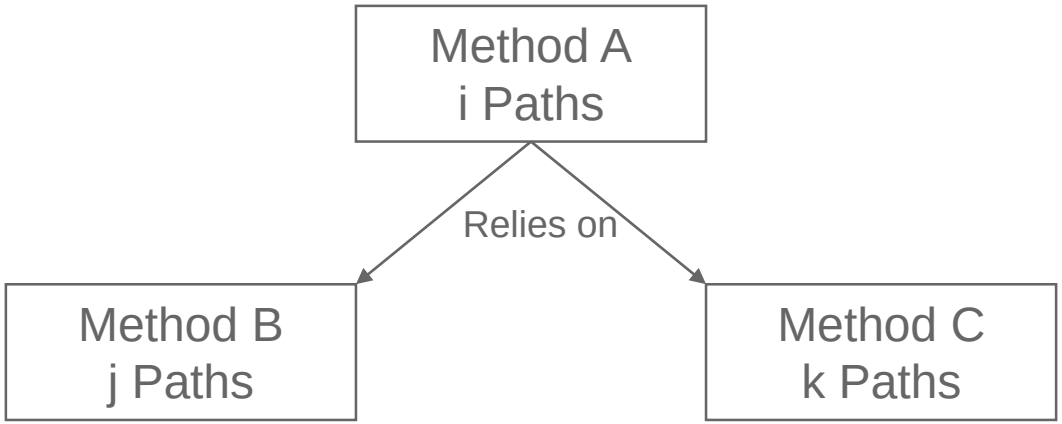
```
public List<Person> getPeople(String surnamePrefix)
{
	DatabaseWrapper db = GetDatabaseWrapper();
	PersonFactory factory = GetPersonFactory();

	List<Person> people = new ArrayList<Person>();

	for(String[] components : db.getPeople(surnamePrefix))
	{
		people.Add(factory.CreatePerson(components);
	}

	return people;
}
```

Unit Testing – Code Design

```
public List<Person> getPeople(String surnamePrefix)
{
        DatabaseWrapper db = GetDatabaseWrapper();
        PersonFactory factory = GetPersonFactory();

        List<Person> people = new ArrayList<Person>();

        for(String[] components : db.getPeople(surnamePrefix))
        {
                people.Add(factory.CreatePerson(components);
        }

        return people;
}
```

Tightly coupled

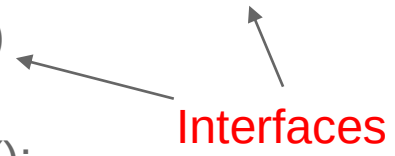
Unit Testing – Isolating Code



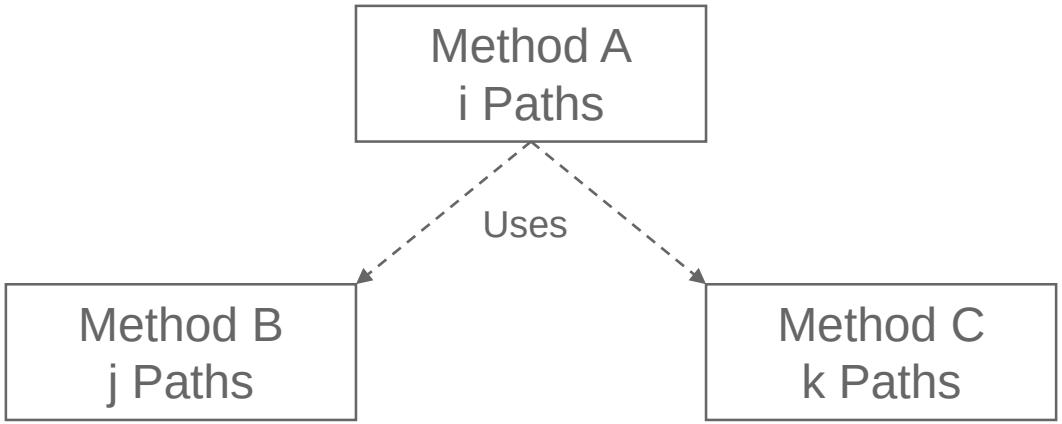
To Test Method A => i * j * k tests

Unit Testing – Code Design

```
public List<Person> getPeople(String surnamePrefix, DatabaseWrapper db,
                              PersonFactory factory)
{
        List<Person> people = new ArrayList<Person>();

        for(String[] components : db.getPeople(surnamePrefix))
        {
                people.Add(factory.CreatePerson(components);
        }

        return people;
}
```



Unit Testing – Isolating Code



To Test Method A => i + j + k tests

Unit Testing – Code Design

```
private DatabaseWrapper db
private PersonFactory factory
```

Injected

```
public List<Person> getPeople(String surnamePrefix)
{
        List<Person> people = new ArrayList<Person>();

        for(String[] components : db.getPeople(surnamePrefix))
        {
                people.Add(factory.CreatePerson(components);
        }

        return people;
}
```

Dependency Injection

- Separates creation of dependencies from behaviour
  - Classes specify dependencies required
  - Framework instantiates & injects services
- Classes have no knowledge of service implementation
- Services can be changed via configuration
  - Theoretically

- E.g. Spring in Java

Unit Tests – Writing the Tests

- Arrange
  - Set up the context for the tests
- Act
  - Call the method being tested
- Assert
  - Check the result of the test

Unit Tests - Arrange

- Instantiate class
- Set context
  - Inject dependencies
  - State of test class
  - Behaviour of dependencies

Unit Testing - Mocking

- Behaviour of Dependencies
  - E.g. Database returns empty list
- Could create/populate DB table
  - Slow & complicated
- Mock object
  - Dummy object that implements interface
  - Can specify behaviour
  - Can verify interaction with mock
- E.g Mockito

Unit Tests - Act

- Test context should be in place
- Call the method to be tested and capture response

Unit Tests - Assert

> Check state of context after test matches expectations

- Value returned by method being tested
- Class containing method being tested
- Dependencies interacted with (called / not called)

Unit Testing – Writing The Tests

```
@Before
public void testSetup()
{
        underTest = new PeopleManager();
        db = mock(DatabaseWrapper.class);
        factory = mock(PersonFactory.class);
}

@Test
public void testGetPeople_DatabaseReturnsEmptyList_ReturnsEmptyList()
{
        when(db.getPeople("Smith")).thenReturn(new ArrayList<Person>());

        List<People> people = underTest.getPeople("Smith", db, factory);

        verify(db, times(1)).getPeople("Smith");
        verify(factory, never()).createPerson(Any);
        assertThat(people.size(), 0);
}
```

Unit Testing – Gotcha's

- Code coverage trap
- No point – code always changing
- Unit tested – don't need to run app
- Need to run tests
  - Run before checking in
  - Run as part of build … broken tests break build

Integration Testing

- Typically performed by developers
  - On local machine
  - Good developers test their own code
- Test interaction between components in system
  - Or a subset of components
- Scale up from unit tests
  - Modules not units
  - ‘Simulate’ dependencies

Smoke Tests

> Check application runs in new environment
  - As fundamental as making sure it starts
  - Check new features are present as expected

> Typically performed by test team
  - Run series of scripted tests to cover functionality

> Can be automated
  - Based on standard scripts
  - Augmented as new functionality added

System Testing

- Test a system from end to end
  - Expensive and time consuming (i * j * k * l * ….)
  - Should be scripted (vs. requirements)
    - i.e. reproducable
- Regression testing for specific changes
- Typically a specific test team
  - Specialised skills

Interacting With Testers

> They are on your side
- Hard to believe sometimes … but should be

> Get them on board with changes
- Communicate how functionality works
- Highlight shortcomings / work-arounds

> Let them help
- Don’t be defensive
- They will often (typically) no more about the system

Interacting With Developers

> Don’t play the blame game
  - Build constructive relationships

> Provide details
  - “It’s broken” doesn’t help

> Listen to the Developers
  - They know how the system is built

(User) Acceptance Testing

> System Test Complete
- i.e. testers and developers happy

> Users test application
- Should not be the first time seen functionality!
- Verify behaviour as expected
- Should be scripted

> Less likely to find bugs
- Functionality issues

Performance Testing

> Load testing/Stress Testing/Scalability Testing
  - How system performs under different loads

> Real-time testing
  - Ensure time constraints are met

Usability Testing

> Test ease of use of application
  - Specialist expertise
  - Many forms
    > Ask people / timings / error tracking

> C.f. User Experience Design
  - Consider ease of use at start

> Accessibility Testing
  - E.g compliance with disability standards

Testing as a Career

> Detail oriented, methodical

> Need to understand domain

- Often better than developer!

A good developer tests their code

Testing – Case Study

- Complex system
  - Many components
  - Fixed API – messages stored as text
- Extreme Programming
  - Pair-programming, Test Driven Development
  - Continuous Integration
- Continuous running of integration tests
  - Design enabled this
  - Production release < 10 minutes

Scott Logic

> We are looking for graduate applications

> Also looking for interns

- Need to have completed penultimate year when doing internship

> Blogs

- More on a range of technical subjects

http://www.scottlogic.com

References

- Cambridge University - http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy_$312_Billion_Per_Year
- Boehm - http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf
- Type Mock - http://www.codeguru.com/blog/category/programming/the-cost-of-bugs.html

SCOTT LOGIC

ALTOGETHER SMARTER