Making the Unstable Stable An Introduction to Testing

CAMERON PRESLEY

@PCAMERONPRESLEY

HTTP://BLOG.THESOFTWAREMENTOR.COM

Pilot. FLYING J.

f(knox)

KNOX HACKS

Outline

- Personal Experience
- Business Case for Testing
- Intro to Testing
- Next Steps
- Additional Resources

Tale of Three Companies

Company A

Large, privately held EMR company

Hundreds of developers

Very mature codebase

Background

Team consisted of 3 developers, 2 QA

Tool that moved patient info from one hospital to another

Errors could lead to data integrity issues

Experience

Fear of refactoring code (too brittle)

Bugs kept coming back

Firefighting mode

No clear focus

What Did I Learn?

Bug count never decreased

Regressions in functionality

Firefighting all the time

Relied upon QA to find issues for us

Company B

Small publicly held company in the healthcare industry

Medical diagnostic device with custom hardware

Coming into a rewrite of the software

Background

Team consisted of 3 developers, 1 QA

Responsible for calculating test results for given samples

Errors could lead to the doctor making the wrong decision

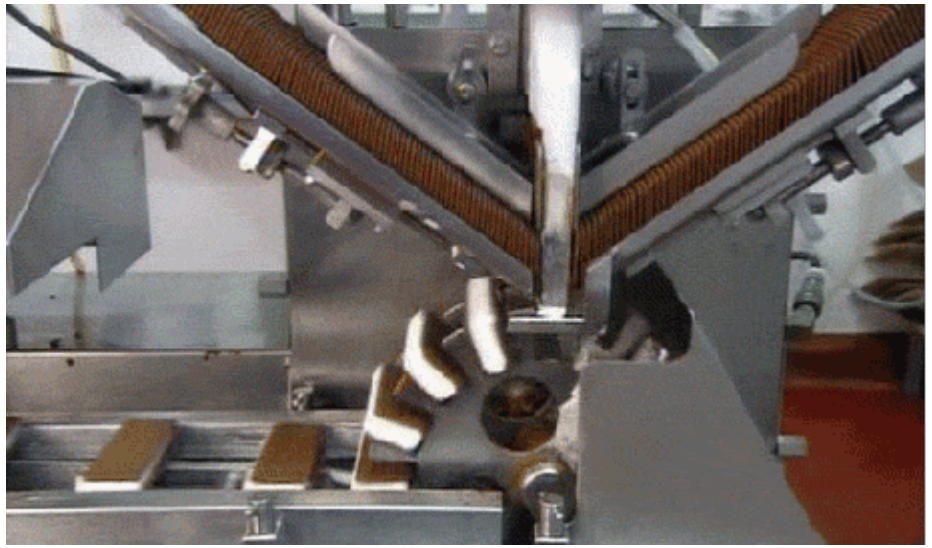
Experience

Refactored constantly

Heavily unit tested components

Bugs never came back

Clear focus on goals



Lessons Learned

Bug count decreased and able to focus on new features

Confidence that we didn’t break anything (able to change freely)

Was it the team or was it testing?

Company C

Startup in the project management space

Help project manager plan “what-if” scenarios

Working on the initial release of the software

Background

Team consisted of 3 developers, 2 QA

Responsible for all parts of the application

Errors could lead to managers making the wrong decision

Experience

Lots of bugs

Demos would crash

- “Pucker factor”

Reactive, not proactive

Experience

Spent time teaching the team how and when to write tests

Product began to stabilize

Fewer bugs were being found

Experience

Shipped a massive feature with **zero** bugs

Confidence in the product soared

Began to focus on new features

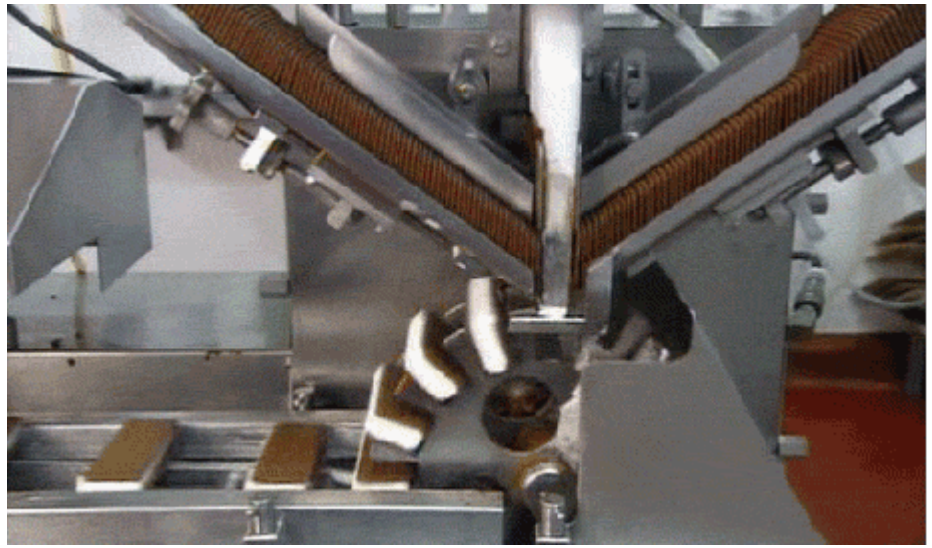
Experience

When I left, there were over 4,000 tests

Entire codebase not covered

- Covered major features/functionality

Never regressed



Personal Experience

Testing allows developers to make the changes in a low-risk fashion

Software becomes easier to change to reflect new requirements

This is pure anecdote, where's the research?

Business Case for Testing

On the Effectiveness of Unit Test Automation at Microsoft

Laurie Williams[1], Gunnar Kudrjavets[2], and Nachiappan Nagappan[2]
[1]*Department of Computer Science, North Carolina State University*
williams@ncsu.edu
[2]*Microsoft Corporation*
{gunnarku, nachin}@microsoft.com

Background

V1 consisted of 1,000 KLOCs written over the course of 3 years

Functional testing (manual testing) was used to ensure the product was working correctly

No automated testing was happening

- Ad-hoc at best

Experiment

V2 consisted of 200 KLOCs of new code and 150 KLOCs of changed code

Mandated unit tests on new functionality

Still relied upon functional testing to ensure no regressions

Results

During code reviews, reviewers could reject changes due to not enough tests

Tests covered 34% of functionality

With manual testing, 85% of functionality was covered

Results

21% drop in bugs at a cost of 30% more development time

Defect Severity	Version 1	Version 2
Severity 1	15.5%	16.8%
Severity 2	49.8%	40.1%
Severity 3	28.7%	18.8%
Severity 4	6.0%	3.4%

Customers Increased 10x

Support Costs Increased 3x

Developers' Perceptions

Spent less time fixing bugs

Writing tests helped with recognizing error conditions

Helped to understand inherited code

Felt more comfortable making changes to code

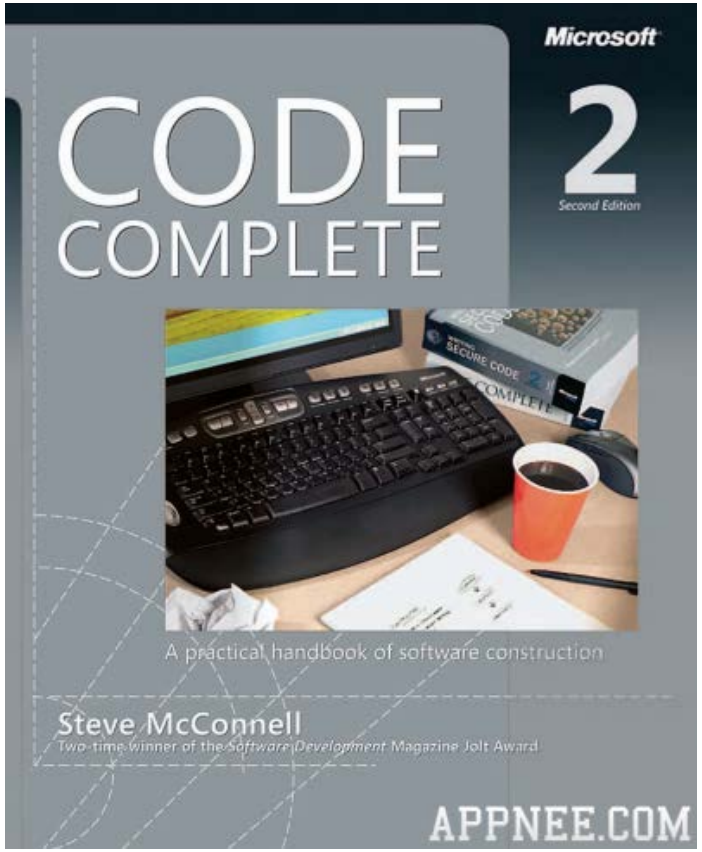
Testers’ Perception

Code is much higher quality

Harder to find bugs, all the obvious ones are gone

Able to spend more time looking for more complicated errors

Not finding “happy path” bugs


Microsoft
CODE
COMPLETE
2
Second Edition
A practical handbook of software construction
Steve McConnell
Two-time winner of the Software Development Magazine Jolt Award
APPNEE.COM

Code Complete

What are the top ways bugs are created?

- Implementation Errors
- Lack of Domain Knowledge
- Changing Requirements
- Communication Mishaps

Code Complete

80% of all bugs come from 20% of the code

80% of maintenance comes from 20% of the code

Code Complete

Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Automated Testing Can Provide…

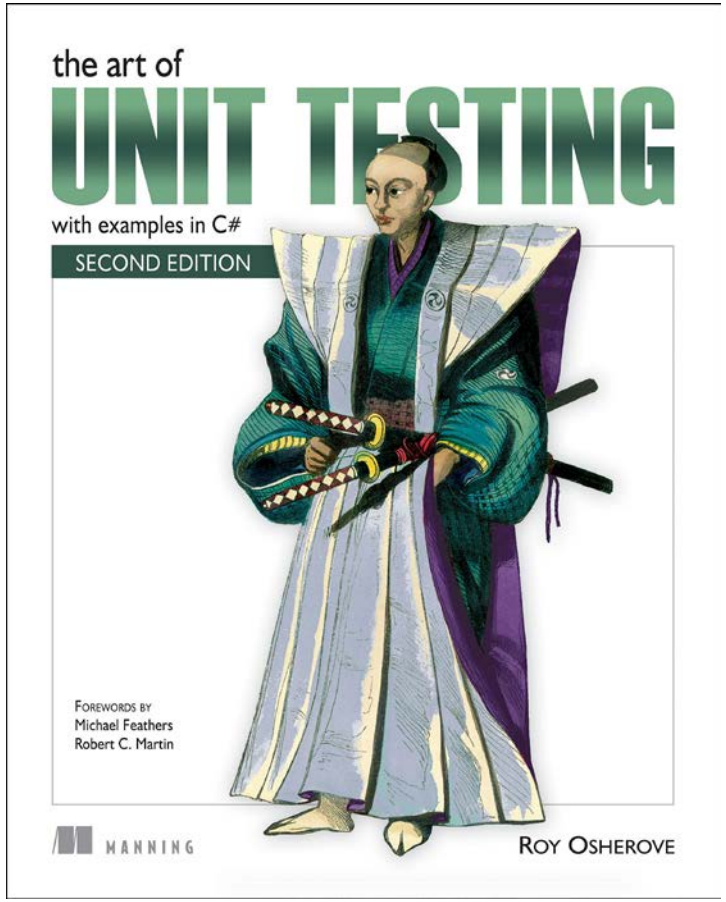
Examples of what the code should do

Built-in documentation

A way to make sure requirements are met

Common language for everyone

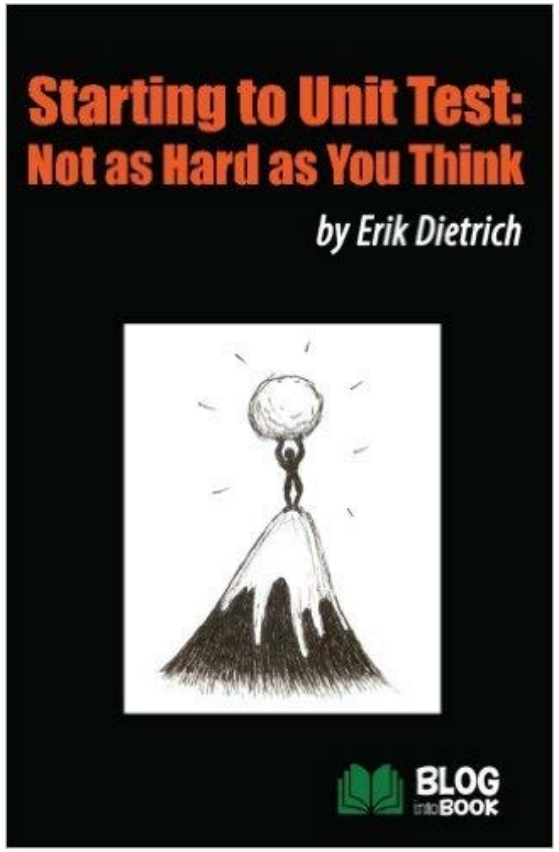
The Art of Unit Testing



The Art of Unit Testing

Stage	Without Tests	With Tests
Implementation	7	14
Integration	7	2
Testing and Bug Fixing		
Testing	3	3
Fixing	3	1
Testing	3	1
Fixing	2	1
Testing	1	1
Total	**26**	**23**
Bugs Found In Production	**71**	**11**

Introduction to Testing



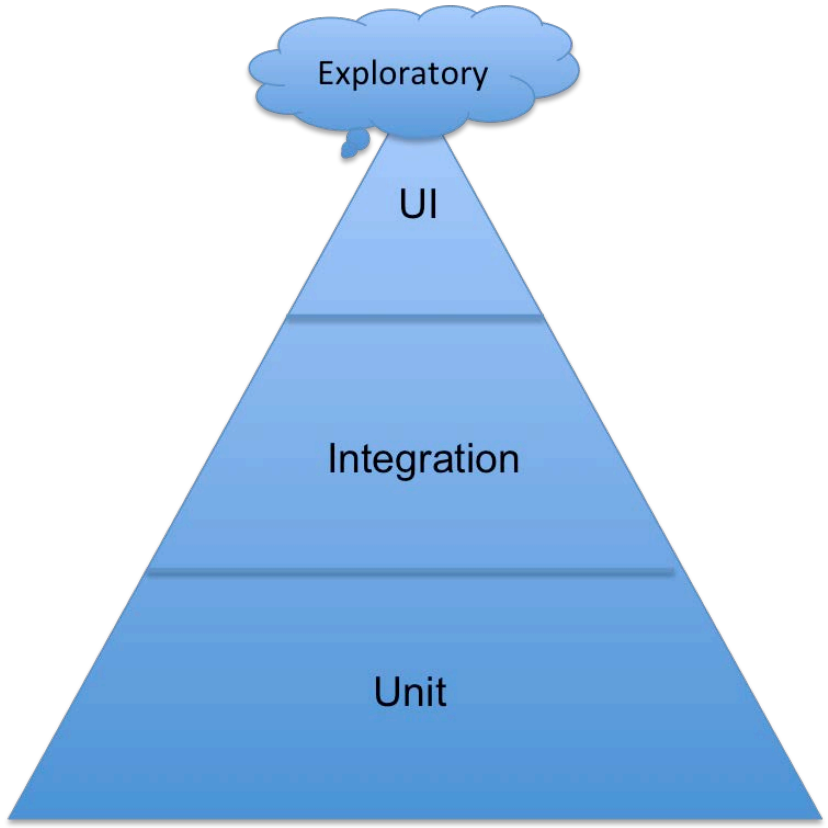
Introducing the Testing Triangle

Introduced by Michael Cohn in

Succeeding with Agile

Breakdown of a test suite

- 10% UI
- 20% Integration
- 70% Unit



Example Test

```
1: [Test]
2: public void And_the_number_is_15_then_FizzBuzz_is_returned()
3: {
4:     var calculator = new FizzBuzzCalculator();
5:
6:     var result = calculator.CalculateFizzBuzz(15);
7:
8:     Assert.AreEqual("FizzBuzz", result);
9: }
```

Parts of a Test (AAA)

Arrange – Get code ready to test
- Create dependencies, set properties, call methods, etc…

Act – Call the code that is being tested
- Method, non-trivial property, constructor, etc…

Assert – Did we get what we expected?
- Throw an exception?, Return an empty list?, Return 42?

Using AAA

```
 1: [Test]
 2: public void And_the_number_is_15_then_FizzBuzz_is_returned()
 3: {
 4:     // Arrange
 5:     var calculator = new FizzBuzzCalculator();
 6:
 7:     // Act
 8:     var result = calculator.CalculateFizzBuzz(15);
 9:
10:     // Assert
11:     Assert.AreEqual("FizzBuzz", result);
12: }
```

Types of Automated Tests

Unit Tests
- Does this one piece work as expected?

Integration Tests
- Does this one piece work as I'd expect with others?

UI/Functional/End-to-End
- When performing this workflow, does the application do what I'd expect?

Unit Tests

"A *unit test* is an automated piece of code that invokes a unit of work being tested, and then checks some assumptions about a single end result of that unit."

- Roy Osherove (*The Art of Unit Testing 2nd Edition*)

Unit Tests

DOES NOT DEPEND UPON EXTERNAL FACTORS!

Typically tests a method or non-trivial property

Super fast to run (100+ per second)

Aren’t likely to change with infrastructure changes

```
 1: [Test]
 2: public void And_the_number_is_15_then_FizzBuzz_is_returned()
 3: {
 4:     // Arrange
 5:     var calculator = new FizzBuzzCalculator();
 6:
 7:     // Act
 8:     var result = calculator.CalculateFizzBuzz(15);
 9:
10:     // Assert
11:     Assert.AreEqual("FizzBuzz", result);
12: }
```

Integration Tests

"*Integration testing* is testing a unit of work without having full control over all of it and using one or more real dependencies, such as time, network, database, threads, number generators, and so on."

- Roy Osherove (*The Art of Unit Testing 2nd Edition*)

Integration Tests

Tests how your code interacts with databases, networks, 3rd party resources

Moderately fast to run (10+ per second)

Decently robust, but can be hard to verify correct behavior

```
 1: [Test]
 2: public void And_the_zip_code_doesnt_have_any_stores_then_an_empty_list_is_returned()
 3: {
 4:   // Arrange
 5:   var storeRetriever = new StoreRetriever();
 6:
 7:   // Act
 8:   var results = storeRetriever.RetrieveStoresFromZip(37886);
 9:
10:   // Assert
11:   CollectionAssert.IsEmpty(results);
12: }
```

UI Tests

Tests the application by driving the UI

Primarily used for end-to-end testing

Very slow (takes seconds per test)

Very fragile since UI changes can cause the test to break

UI Tests

Record and play-back

- Records the interaction between user and application
- Can be done with QA

Coded

- Specify UI controls by ID or label to interact with
- Primarily developer driven

Tools of the Trade

Testing Framework

Absolute minimum for writing tests

Provides a way to mark classes/methods as tests

Allows us to check if result is what we expected

- AreEqual, IsTrue, IsFalse, etc…

Testing Framework

```
1: using NUnit.Framework;
2:
3: [TestFixture]
4: public class When_calculating_Fizz_Buzz_numbers
5: {
6:   [Test]
7:   public void And_the_input_is_15_then_FizzBuzz_is_returned()
8:   {
9:     // Arrange
10:    var calculator = new FizzBuzzCalculator();
11:
12:    // Act
13:    var result = calculator.CalculateFizzBuzz(15);
14:
15:    // Assert
16:    Assert.AreEqual("FizzBuzz", result);
17:  }
18: }
```

Mocking Framework

Unit tests are faster than integration tests

Lot of code involves different classes collaborating together

How do we unit test these classes?

Create test dependencies that we can control
- Using interfaces or inheritance

Mocking Framework

Creates a lot of test code

Mocking framework uses reflection to create dependencies

Can be used to check that methods/properties were called or to stub a response

Examples

- NSbustitute, Moq, Mockito

Mocking Framework
Asserting Methods Were Called

```
 1: using NUnit.Framework;
 2: using NSubstitute;
 3:
 4: [TestFixture]
 5: public class When_displaying_the_dashboard
 6: {
 7:   [Test]
 8:   public void Then_the_storeRetriever_is_called_with_the_correct_zip_code()
 9:   {
10:     // Arrange
11:     var storeRetriever = Substitute.For<IStoreRetriever>();
12:     var presenter = new DashboardPresenter(storeRetriever);
13:     presenter.ZipCode = "37934";
14:
15:     // Act
16:     presenter.DisplayDashboard();
17:
18:     // Assert
19:     storeRetriever.Received(1).RetrieveStores("37934");
20:   }
21: }
```

Mocking Framework Stubbing Responses

```
 1: using NUnit.Framework;
 2: using NSubstitute;
 3:
 4: [TestFixture]
 5: public class When_displaying_the_dashboard
 6: {
 7:   [Test]
 8:   public void And_there_are_no_results_then_the_dashboard_shows_the_correct_message()
 9:   {
10:     // Arrange
11:     var storeRetriever = Substitute.For<IStoreRetriever>();
12:     storeRetriever.RetrieveStores(null).ReturnsForAnyArgs(new List<Store>());
13:     var presenter = new DashboardPresenter(storeRetriever);
14:
15:     // Act
16:     presenter.DisplayDashboard();
17:
18:     // Assert
19:     Assert.AreEqual("There are no results.", presenter.Message);
20:   }
21: }
```

UI Framework

Provides a mechanism to interact with UI controls

Great to provide end-to-end workflow testing

Examples

- Test Stack White (Desktop), Selenium (Web), HP Quick Test Pro

Tools of the Trade

Testing Framework
- Allows us to write tests

Mocking Framework
- Allows us to switch out real dependencies for fake ones

UI Framework
- Allows us to interact with the UI components

Next Steps

Learning How to Write Tests

Sample Exercises

- Fizz Buzz
- String Calculator
- Bowling Game
- Mars Rover

Next Steps
Identify Problem Areas

Find places in your codebase that have few dependencies
- Easier to test and will help build confidence

Find places in your codebase that generates bugs
- Remember 80% of issues come from 20% of the code
- Might need to refactor code to make it testable

Don’t know the areas? Ask QA

Next Steps
Low Impact Testing

Don’t have to immediately write tests around everything

Start by adding tests on code that you change going forward

Just focus on bug fixes

- Write a test that fails because the bug exists
- Test should pass once the bug is fixed

Next Steps
Convincing Your Boss

Goal is to move applications to production

Remember, bugs are more expensive the later they're found
- 25x more expensive if not caught before production

Every time QA finds a bug, there's going to be churn

Puts more pressure on QA and the developers

If You're a Manager…

Identify those on your team that know how to write tests

- Have those developers distribute their knowledge to others

Provide training on how to write tests

- Pluralsight has some great resources (free 12 month access for all MSDN holders)
- *The Art of Unit Testing (2nd Edition)* by Roy Osherove

Incorporate slack time to give developers time to learn

Remember!

Goal is to decrease time to production, not 100% code coverage

Not all applications need tests

Not every single line of code needs coverage

- 80% of issues come from 20% of the code

Be pragmatic, not dogmatic with testing

Want to Learn More?

10,000 foot view on testing

- *Starting to Unit Test: Not as Hard as You Think* (e-book) by Erik Dietrich

In-Depth Look at Testing

- *The Art of Unit Testing (2nd Edition)* by Roy Osherove

Testing Legacy Code

- *Working Effectively With Legacy Code* by Michael Feathers
- *Refactoring* by Martin Fowler

Additional Resources

Books

- *Code Complete (2nd Edition)* by Steve McConnell

Articles

- *On the Effectiveness of Unit Testing at Microsoft*

Presentations

- Unit Testing Makes Me Faster by Jeremy Clark

Videos

- Introduction to .NET Testing with NUnit by Jason Roberts

Thanks!

Email: Cameron@TheSoftwareMentor.com

Twitter: @PCameronPresley

Blog: http://blog.TheSoftwareMentor.com